

- 以配置和使用最广泛的Bash脚本为对象进行讲解
- 结合了多位Linux系统开发人员和维护人员对Shell脚本编程的实践经验，对相关知识点以及示例进行梳理，具有很强的实用性
- 较系统地提供了Shell编程的基础知识和技巧，读者可以根据自己的需要选择全部学习还是只学习部分内容
- 内容精炼，例子详尽，讲解由浅入深，方便读者入门学习，也可作为手头工具书随时查阅



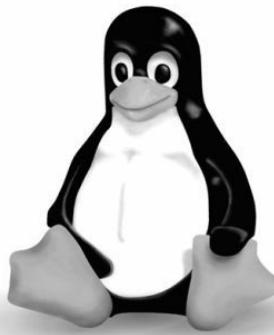
Linux Bash 编程与脚本应用实战

· 马玉军 赫军 编著 ·



本书资源下载

清华大学出版社



Linux Bash

编程与脚本应用实战

· 马玉军 郝军 编著 ·

清华大学出版社
北京

内容简介

本书从实际的应用场景出发，深入浅出地介绍了编写Shell脚本所包含的各项常用技术，使读者在不知不觉中就能掌握Shell脚本的编写和使用技巧，是Shell脚本编程开发初学者的绝佳首选。

本书分为18章，分别介绍了Linux系统基本知识，Shell编程基础，Shell编程的常用语法知识，如变量、特殊符号、文件处理、分支结构和循环结构、正则表达式、grep命令、sed编辑器、gawk编辑器等，如何对Shell脚本的执行进行控制和优化，最后通过两个应用实例，加深读者对Linux系统下Shell编程的认识，提高读者的编程能力。

本书适用于Shell编程初学者和Linux系统管理的初学者，可以作为日常学习的教材，还可以作为日常管理的参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，翻印必究。举报电话：**010-62782989 13501256678**
13801310933

图书在版编目（CIP）数据

Linux Bash编程与脚本应用实战/马玉军，郝军编著. —北京：清华大学出版社，2015

ISBN 978-7-302-38446-5

I . ①L... II . ①马...②郝... III. ①Linux操作系统—程序设计
IV. ①TP316.89

中国版本图书馆CIP数据核字（2014）第260912号

责任编辑：夏非彼

责任校对：闫秀华

责任印制：

出版发行：清华大学出版社

<http://www.tup.com.cn>

地 址：北京清华大学学研大厦A座

邮 编：100084

社总机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者：清华大学印刷厂

经 销：全国新华书店

开 本：190mm×260mm

印 张：21

字 数：538千字

版 次：2015年1月第1版

印 次：2015年1月第1次印刷

印 数：1~3000

定 价：49.00元

产品编号：055067-01

前言

为何选择**Bash Shell**?

Linux系统的发布版本目前有很多，而Debian系列的Ubuntu系统是使用率较高的Linux系统，在Ubuntu系统中默认安装的Bash也是所有Linux系统中使用率较高的脚本语言，因此本书采用Linux Bash来讲解脚本语言编程与应用。

本书真的适合你吗？

本书旨在帮助那些刚接触Linux系统以及Shell编程的“新人”，提供Shell编程中的常用语法以及使用率较高的知识；本书涉及Shell编程中的变量使用、条件结构、循环结构、函数、正则表达式、grep命令、sed编辑器、gawk编辑器的常用方式，读者可以根据自己的需要选择全部学习或只学习部分内容。

本书是一本全面的Shell脚本编程技术书，是比较实用的Shell编程案例书，部分实例和框架在下面给读者做一个预览。

涉及的示例和案例

使用Shell操作MySQL数据库

系统内存监控

系统硬盘空间监控

进程空闲监控

日志定时备份

计算器模拟

后台程序运行过程控制

前台程序暂停及继续执行

使用键盘发送信号控制进程的执行

正则表达式的使用

gawk编辑器的使用

sed编辑器的使用

算术运算的实现

函数的递归调用

函数的嵌套使用

数组在函数中的应用

循环结构的使用

循环结构的控制

条件结构的使用

测试命令的使用

设备文件的挂载和卸载

输入输出的重定向

引号的使用

元字符的使用

通配符的使用

管道的使用

位置变量的使用

环境变量的使用

本书特点

- 本书介绍的知识以及实例来自于多位“奋斗”在Linux系统中的开发人员和实施维护人员，结合众人多年对Shell脚本编程的了解和独特见解，因此本书的实用性是非常强的。书中所述内容不管是对理论知识的介绍，还是具体实例的开发，都是从实际应用角度出发，精心选择在实际项目中使用的实例进行细致的讲解和分析。
- 深入浅出、轻松易学，以实例为主线，激发读者的阅读兴趣，让读者能够真正学习到在Linux系统下如何使用Shell脚本解决

实际问题。

- 贴近读者、贴近实际，本书中的实例大部分都紧跟一个知识点，做到理论和实际相结合，用大量的实例说明如何使用讲述的知识来编写Shell脚本，从而解决实际问题。

贴心提醒

本书根据在实际编写Shell脚本的过程中出现的种种问题，在各章使用了很多“注意”等小提示，让读者可以在学习过程中更轻松地理解相关知识点及概念。

面向读者

- Linux系统Shell编程初学者
- Linux环境开发人员
- Linux系统管理员
- 网络管理员
- 服务器管理员

资源下载

本书配套示例脚本包下载地址（请注意字母大小写和数字）如下：

<http://pan.baidu.com/s/1bnER5SF>

如果下载有问题，请发邮件到电子邮箱booksaga@163. com，邮件

标题为“求bash脚本”。

本书第1~15章由南阳理工学院的马玉军编写，第16~18章由郝军编写，其他参与编写的人员还有李光明、史帅、庞国威、娄晓东、郭刚、李兰英、林龙、张泽娜、李春城、宋楠、李为民、祝明慧、周瑞、潘承恩、殷龙。

编者

2014年9月

目 录

[前言](#)

[第1章 学习Shell的背景知识——Linux系统](#)

[1.1 Linux系统基础](#)

[1.1.1 Linux系统的发展](#)

[1.1.2 Linux系统和Windows系统的区别](#)

[1.1.3 Linux的启动过程](#)

[1.2 Linux文件系统基础介绍](#)

[1.2.1 必须了解的节点inode](#)

[1.2.2 Linux系统文件结构](#)

[1.3 学会Linux系统的基本使用](#)

[1.3.1 系统的登录与退出](#)

[1.3.2 系统基本选项配置](#)

[1.3.3 文本编辑器的使用](#)

[1.4 走进Shell](#)

[1.4.1 什么是Shell](#)

[1.4.2 Shell在Linux系统中的作用](#)

[1.4.3 Shell的种类](#)

[1.5 小结](#)

[第2章 迈出Shell脚本编程第一步](#)

[2.1 Shell脚本语言概述](#)

[2.1.1 Shell脚本语言的定义](#)

[2.1.2 Shell终端的基本使用](#)

[2.1.3 Shell终端菜单的使用](#)

[2.2 Shell命令格式介绍](#)

[2.2.1 Shell命令格式](#)

[2.2.2 命令行界面介绍](#)

[2.2.3 如何获取帮助](#)

[2.3 第一个Shell程序：Hello, Bash Shell！](#)

[2.3.1 创建Shell脚本](#)

[2.3.2 Shell脚本中的格式介绍](#)

[2.3.3 如何执行Shell程序](#)

[2.4 小结](#)

[第3章 Shell常用命令大演练](#)

[3.1 Shell命令使用基础](#)

[3.1.1 文件类型](#)

[3.1.2 绝对路径和相对路径](#)

[3.1.3 文件属性和文件权限](#)

[3.1.4 用户和用户组](#)

[3.1.5 特殊目录介绍](#)

[3.2 用户和用户组管理](#)

[3.2.1 用户管理常用命令](#)

[3.2.2 用户组管理常用命令](#)

[3.2.3 其他常用命令](#)

[3.3 文件和目录操作](#)

[3.3.1 文件操作常用命令](#)

[3.3.2 目录操作常用命令](#)

[3.3.3 文件权限管理常用命令](#)

[3.3.4 查找文件常用命令](#)

[3.4 系统管理相关](#)

[3.4.1 网络操作常用命令](#)

[3.4.2 系统资源管理常用命令](#)

3.4.3 磁盘信息查看常用命令

3.5 小结

第4章 Shell脚本中的变量

4.1 变量的简单使用

4.1.1 变量的使用

4.1.2 变量的输入和输出

4.1.3 变量的输出命令echo

4.2 Shell中特殊变量的使用

4.2.1 位置参数介绍

4.2.2 \${@}和\${*}的区别

4.3 环境变量的使用

4.3.1 Shell中的环境变量

4.3.2 环境变量的配置文件

4.3.3 全局环境变量和本地环境变量

4.3.4 环境变量的设定

4.3.5 环境变量的取消

4.4 小结

第5章 Shell脚本中的特殊符号

5.1 引号的使用

5.1.1 单引号的使用

5.1.2 双引号的使用

5.1.3 倒引号的使用

5.2 通配符和元字符

5.2.1 使用通配符

5.2.2 使用元字符

5.3 管道

5.4 其他特殊字符介绍

[5.4.1 后台运行符](#)

[5.4.2 括号](#)

[5.4.3 分号](#)

[5.5 小结](#)

[第6章 Linux中的文件处理](#)

[6.1 Linux中的文件类型](#)

[6.1.1 设备文件](#)

[6.1.2 设备文件的挂载和卸载](#)

[6.1.3 链接文件](#)

[6.1.4 文件描述符](#)

[6.2 标准输入、输出与错误](#)

[6.2.1 标准输入](#)

[6.2.2 标准输出和标准错误](#)

[6.3 重定向](#)

[6.3.1 重定向的定义](#)

[6.3.2 输入重定向](#)

[6.3.3 输出重定向](#)

[6.4 合并标准输出和标准错误](#)

[6.5 小结](#)

[第7章 Shell脚本中的分支结构](#)

[7.1 测试命令的使用](#)

[7.1.1 测试命令的基础结构](#)

[7.1.2 测试文件类型](#)

[7.1.3 测试字符串](#)

[7.1.4 测试数值](#)

[7.1.5 复合测试条件](#)

[7.2 if分支结构](#)

[7.2.1 if-then结构](#)
[7.2.2 if-then-else结构](#)
[7.2.3 嵌套结构](#)
[7.3 case多条件分支结构](#)

[7.3.1 case结构基础](#)
[7.3.2 在Shell脚本中使用case结构](#)
[7.3.3 select命令的使用](#)

[7.4 小结](#)

[第8章 Shell中的循环结构](#)

[8.1 for循环](#)
[8.1.1 使用for-in结构](#)
[8.1.2 C式for结构](#)
[8.2 while命令的使用](#)
[8.2.1 使用while结构](#)
[8.2.2 多条件的while结构](#)
[8.2.3 使用until命令](#)

[8.3 命令的嵌套](#)
[8.3.1 for命令的嵌套](#)
[8.3.2 while命令的嵌套](#)
[8.3.3 until命令的嵌套](#)
[8.4 循环控制符](#)

[8.4.1 使用break中断](#)
[8.4.2 使用continue继续](#)

[8.5 小结](#)

[第9章 Shell中的函数](#)
[9.1 函数的基本用法](#)
[9.1.1 函数的创建与使用](#)

[9.1.2 函数的参数](#)

[9.2 函数的返回值](#)

[9.2.1 返回值基础](#)

[9.2.2 函数的默认返回值](#)

[9.2.3 return命令的使用](#)

[9.2.4 使用函数的返回值](#)

[9.3 函数中的全局变量和局部变量](#)

[9.3.1 全局变量](#)

[9.3.2 局部变量](#)

[9.4 数组与函数](#)

[9.4.1 数组作为函数参数](#)

[9.4.2 数组作为函数返回值](#)

[9.5 脚本函数递归](#)

[9.6 函数的嵌套调用](#)

[9.7 小结](#)

[第10章 Shell脚本编写技巧](#)

[10.1 脚本编写规范](#)

[10.1.1 命名规范](#)

[10.1.2 注释风格](#)

[10.1.3 其他需要注意的规范](#)

[10.2 脚本优化](#)

[10.2.1 Shell脚本优化原则](#)

[10.2.2 提供足够的提示信息](#)

[10.3 脚本使用技巧](#)

[10.3.1 命令选项说明](#)

[10.3.2 算术运算](#)

[10.3.3 高级算术运算](#)

10.4 小结

第11章 正则表达式

11.1 正则表达式基础

11.1.1 正则表达式的定义

11.1.2 正则表达式的分类

11.2 基本正则表达式的常用符号

11.2.1 使用点字符匹配单字符

11.2.2 使用定位符

11.2.3 使用“*”匹配字符串中的单字符或其重复序列

11.2.4 使用“\”屏蔽一个特殊字符的含义

11.3 扩展正则表达式的常用符号

11.3.1 使用“[]”匹配一个范围或集合

11.3.2 使用“{\}”匹配模式结果出现的次数

11.3.3 问号的使用

11.4 小结

第12章 Shell中的文本搜索工具——grep家族

12.1 grep的基础使用

12.1.1 grep命令的基本使用方式

12.1.2 grep选项

12.1.3 行数

12.1.4 大小写敏感

12.1.5 显示非匹配行

12.1.6 查询多个文件或多个关键字

12.2 grep和正则表达式

12.2.1 模式范围以及范围组合

12.2.2 定位符的使用

12.2.3 字符匹配

[12.2.4 模式出现几率](#)

[12.2.5 匹配特殊字符](#)

[12.3 grep命令的扩展使用](#)

[12.3.1 类名的使用](#)

[12.3.2 egrep命令的使用](#)

[12.3.3 fgrep命令的使用](#)

[12.4 grep命令使用实例](#)

[12.4.1 目录搜索——查找特定目录或文字](#)

[12.4.2 使用ps命令检索特定的进程](#)

[12.5 小结](#)

[第13章 sed编程](#)

[13.1 认识sed](#)

[13.1.1 sed工作模式](#)

[13.1.2 sed常用指令](#)

[13.1.3 sed常用选项](#)

[13.1.4 sed地址范围](#)

[13.2 sed编辑器常用命令](#)

[13.2.1 替换命令的使用](#)

[13.2.2 删除命令的使用](#)

[13.2.3 文本的添加和替换](#)

[13.3 高级sed编程](#)

[13.3.1 同时处理多行数据](#)

[13.3.2 sed编辑器的空间](#)

[13.3.3 sed编辑器的反向](#)

[13.3.4 重定向sed的输出](#)

[13.4 小结](#)

[第14章 gawk编程](#)

14.1 gawk概述

14.1.1 gawk基本介绍

14.1.2 gawk基本使用

14.2 变量的使用

14.2.1 内置变量的使用

14.2.2 自定义变量的使用

14.2.3 数组的使用

14.3 结构的使用

14.3.1 条件结构的使用

14.3.2 循环结构的使用

14.3.3 循环结构控制语句

14.4 函数的使用

14.4.1 算术函数的使用

14.4.2 字符串处理函数的使用

14.4.3 时间函数的使用

14.5 小结

第15章 脚本控制

15.1 Linux信号控制

15.1.1 Linux信号机制简介

15.1.2 使用Shell脚本操作信号

15.2 进程的控制

15.2.1 后台运行符介绍

15.2.2 运行进程的控制

15.2.3 nohup命令的使用

15.3 脚本运行的优先级

15.3.1 优先级介绍

15.3.2 使用nice指定优先级

15.3.3 使用renice重置优先级

15.4 小结

第16章 脚本运行的优化

16.1 添加窗口

16.1.1 dialog软件的安装

16.1.2 dialog命令的帮助选项

16.1.3 dialog命令的使用

16.2 dialog常用窗口的使用

16.2.1 消息窗口

16.2.2 yesno窗口

16.2.3 文本框的使用

16.2.4 菜单的使用

16.3 颜色的使用

16.4 创建菜单

16.4.1 在Shell脚本中创建菜单

16.4.2 创建子菜单函数

16.4.3 脚本的整合

16.5 小结

第17章 Shell实战之系统管理

17.1 系统监测

17.1.1 系统监控基础

17.1.2 Ubuntu自带的系统监控工具

17.1.3 监控脚本的编写

17.2 计划任务的实现

17.2.1 at命令的使用

17.2.2 atq命令的使用

17.2.3 cron的使用

17.3 网络管理

17.3.1 网络配置

17.3.2 服务器的安装

17.4 日志管理

17.4.1 日志简介

17.4.2 守护进程syslogd

17.4.3 日志的备份操作

17.4.4 日志的定时操作

17.5 小结

第18章 Shell实战之数据库操作

18.1 Linux系统中的数据库

18.1.1 SQLite简介

18.1.2 SQLite的图形化操作

18.1.3 MySQL简介

18.2 SQL语句

18.2.1 SQL语言基本介绍

18.2.2 基本的SQL操作

18.2.3 在Shell脚本中执行SQL语句

18.3 图书管理系统中数据库操作实例

18.3.1 数据库操作基本流程

18.3.2 创建表

18.3.3 增加图书信息

18.3.4 修改图书信息

18.3.5 删除图书信息

18.4 小结

第1章 学习Shell的背景知识——Linux系统

Shell脚本的编写依赖于Linux系统，因为脚本的编写、执行、修改都需要在Linux系统之上。所以在正式介绍Shell脚本的相关内容之前，首先介绍Linux的基础知识，从而为后面讲解如何编写Shell脚本打下基础。

本章的主要内容如下：

- Linux系统基础介绍
- Linux系统的文件系统介绍
- Linux系统的基本使用
- Shell的基本介绍

1.1 Linux系统基础

Linux系统最初源于Unix系统，是一套 Unix-like 的作业系统，也是 Unix 的一种，Linux系统是一类Unix计算机操作系统的统称。Linux操作系统也是自由软件和开放源代码发展中最著名的例子，发展到现在已经经历了二十余年，下面将简要介绍Linux系统的发展以及它与Windows 系统的区别。

1.1.1 Linux系统的发展

对于Windows系统来说，其核心代码是不对外公布的。这样虽然对于系统的保密性和安全性起到了一定的作用，但从长远来说，不利于技术的发展和进步。1983年，理察·马修·斯托曼（Richard Stallman）创立了GNU计划（GNU Project）。这个计划的目标是发展一个完全免费自由的Unix-like操作系统，从而打破操作系统的源码只在少数人手中的限制，打破软件技术发展的瓶颈。

1991年4月，芬兰赫尔辛基大学学生林纳斯·托瓦兹（Linus Torvalds）由于不满意Minix这个教学用的操作系统，出于个人爱好，开发出了Linux系统的第一版Linux 0.01。此时的Linux系统还需要运行在Minix系统之下，林纳斯宣布该系统可以被任意地下载和修改，希望大家能够对这个系统进行完善。许多专业程序员加入其中，他们自愿地开发Linux系统的应用程序，并借助Internet拿出来让大家一起修改，所以它周边的程序越来越多，Linux本身也逐渐发展壮大起来。在所有开发人员的共同努力下，GNU组件可以运行于Linux内核之上。直到1994年3月，Linux 1.0版正式发布，自此，Linux系统以崭新的形式投入到市场中。

Linux系统是一种free的操作系统，这种free不单单指可以自由地从互联网上传、下载Linux操作系统，对于大多数人来说，free最重要的含义是Linux系统是一种自由软件，用户可以自由地修改Linux系统的源代码，从而满足自己的实际需要。当修改完以后，还可以将修改的内容上传到互联网上，供其他人员学习或修改。这样既提高了个人的能力，也帮助了Linux系统修复BUG和进行系统的更新，从而提高了Linux系统的稳定性和执行效率。

Linux系统发展至今存在很多的发行版本。在每一种发行版本中都包括Linux内核、必备的GNU程序库和工具、命令行Shell、图形界面的

X Window系统和相应的桌面环境，并且还有数量众多的应用软件和应用程序。应用范围比较广泛的发行版本如下：

- Ubuntu
- Red Hat
- Suse
- Fedora Core
- Red Flag（红旗）

在上面的Linux系统发行版本中，每一种发行版本都可以独立使用。用户可以根据自己的需要来选择使用相应的版本。

1.1.2 Linux系统和Windows系统的区别

在日常工作和生活中，一般使用的操作系统为Windows操作系统，Linux系统与Windows的主要区别是Linux是自由的软件。

Linux对硬件要求较低，而Windows要求较高。Windows一般都要求硬件有最低配置，这个最低配置仅是保证Windows系统能够正常运行。而对于Linux系统来说，保证Windows系统正常运行的最低配置就是可以使得Linux系统运行得非常顺畅，即使增加几个应用程序，也同样可以正常使用。

Linux系统和Windows系统都提供了较好的网络功能，但是在安全与稳定方面却存在着差异。Linux提供对当前TCP/IP协议的完全支持，并且包括对下一代Internet协议IPv6的支持。Linux内核还包括了IP防火

墙代码、IP防伪、IP服务质量控制及许多安全特性。因此，其网络性能要优于Windows系统。

微软的Windows产品是专门为桌面系统设计的，面向中、低端用户，友好的界面使其在计算机操作系统市场占据了较高的市场份额。虽然Linux有两个不错的图形操作界面（GNOME和KDE），也能提供不错的桌面服务，但不及Windows系统方便灵活，这严重影响了它的普及率。

由于源代码开放的特点，成千上万的人可以开发Linux，快速找到并修改其错误码，而且许多硬件厂商也可以直接阅读其源代码，从而迅速提供对它的支持。

1.1.3 Linux的启动过程

Linux系统的启动过程是指从用户开机、通电到可以输入用户名和密码登录系统的这段过程，当计算机执行这段过程时，所有的信息都在屏幕上一闪而过，并且会出现很多的信息。因此对于初学者来说，Linux的系统启动过程会显得非常“神秘”，但是实际上Linux的启动并不是想象中的那么复杂。Linux系统的启动过程如图1-1所示。

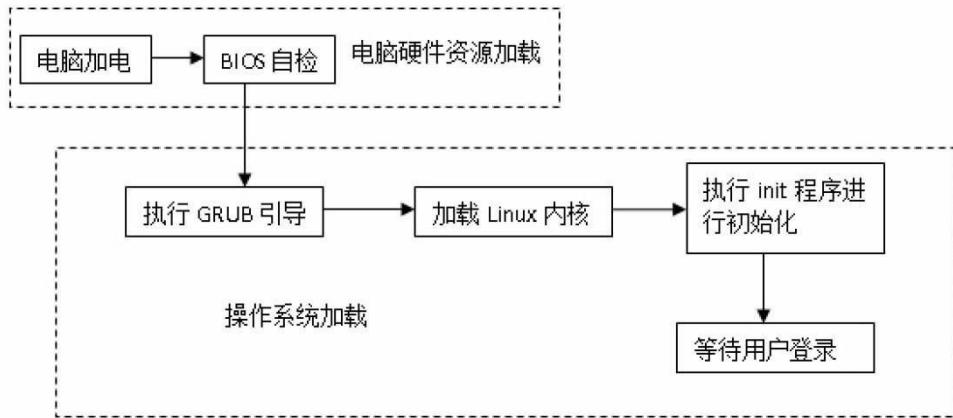


图1-1 Linux系统启动示意图

当电脑开机后，首先会启动BIOS程序进行系统自检。当自检通过后就会进入到硬盘的MBR（主引导记录）中执行存储在该区域的程序。这个程序就是Linux系统中的grub程序（系统引导程序），grub程序的主要作用就是确定存储在硬盘中的操作系统从什么地方启动，配置文件/boot/grub/grub.conf存储了在硬盘中记载的所有的操作系统以及操作系统的存储信息，该文件的内容如下：

```

# DO NOT EDIT THIS FILE
#
# It is automatically generated by /usr/sbin/grub-mkconfig using
# information from /etc/grub.d and settings from /etc/default/grub
#
#### BEGIN /etc/grub.d/00_header ####
if [ -s $prefix/grubenv ]; then
    load_env
fi

```

```
set default="0"

if [ ${prev_saved_entry} ]; then
    set saved_entry=${prev_saved_entry}
    save_env saved_entry
    set prev_saved_entry=
    save_env prev_saved_entry
    set boot_once=true
fi

function savedefault {
    if [ -z ${boot_once} ]; then
        saved_entry=${chosen}
        save_env saved_entry
    fi
}

.....//省略部分文件内容

### BEGIN /etc/grub.d/40_custom ###
# This file provides an easy way to add custom menu entries.
# menu entries you want to add after this comment. Be careful!
# the 'exec tail' line above.

### END /etc/grub.d/40_custom ###
```

在配置文件/boot/grub/grub.conf中，语句default是指默认使用哪个操作系统，所有的操作系统都在Grub画面中显示。在启动系统时，Grub将

直接体现为操作系统选择菜单。在系统启动时，会出现一个操作系统选择菜单，用户可以根据需要选择需要启动哪个。其基本内容如图1-2所示。

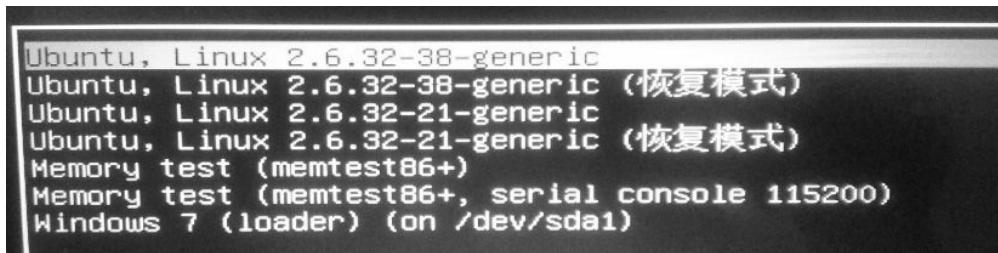


图1-2 操作系统选择菜单

对于图1-2中的操作系统选择菜单来说，光标所在的位置就是马上要进行启动的操作系统。在实际使用时，可以根据实际需要使用上下箭头来选择启动哪个系统。而如果在默认的时间中没有选择，系统自动选择两条所在的位置所指示的系统进行启动。该默认时间也可以在配置文件grub.conf中进行修改。

图1-2中所示的操作系统选择菜单仅是在硬盘中包含多个操作系统时才会出现。而且该菜单显示的内容和本机上安装的是哪些系统有关系，因此，其现实的内容不会都是一样的。一般情况下，操作系统选择菜单中显示的内容和配置文件grub.conf中的title后面的内容相同，存在多少个title，那么在选择菜单中就出现多少个选项。如果只存在一个操作系统，那么在操作系统选择菜单中仅显示一条选项。在配置文件grub.conf中设定default的数值是多少，那么默认的操作系统就是哪一个。

注意

default的初始值为0，而不是1，因此在确定默认的操作系统时

要注意选项的位置为default值再加1。

配置文件中title开始的部分的下一行为确定操作系统存储位置。在Linux系统中，使用hd表示硬盘，紧跟其后的数字表示第几块硬盘。逗号后面的数字则表示该硬盘的第几个分区。如语句root (hd0, 0) 就表示第一块硬盘的第一个分区。括号中的数字和default的数值是一一对应的。这就可以知道是启动的是哪一个操作系统。

紧跟其后的kernel指代存放Linux内核的位置。（hd0,0）指代boot目录，所以这个文件在boot目录中。而语句ro root=LABEL=/就是设置根目录的位置，root前面的ro表示read only，所以有这行的设定，才能读取根目录。

在选择了合适的操作系统以后就启动init服务对操作系统进行初始化。启动init服务实际上就是执行init程序，该程序是Linux系统运行的第一个程序，也是在Linux系统中运行的第一个进程，该程序根据启动模式（即run_level）确定如何启动系统。

在进行系统的启动时，Ubuntu中的init进程并不会直接去目录文件/etc/init.d或者/etc/rc\${runlevel}.d中查找相应的文件并且执行，而是采用了折衷的办法，通过/etc/init下的相应配置文件来调用/etc/rc\${runlevel}.d中的脚本以启动采用旧式System V-style的服务。对于Ubuntu系统来说，其启动文件存在于以下3个目录中：

- /etc/init
- /etc/init.d
- /etc/rc\${runlevel}.d

在启动系统时，首先判断启动模式（即run_level），即使用何种方式启动Linux系统。/etc/init.d/rc被调用了，并且传入了早前设置好的系统runlevel作为参数。而/etc/init.d/rc会根据传入的runlevel参数调用/etc/rc\${runlevel}.d/下的脚本（以S开头）以启动服务。

对于Ubuntu系统来说，启动模式run_level一般分为7种，其相应的作用如表1-1所示。在一般情况下，采用的run_level为1、3、5这3种启动模式。对于启动模式0和6来说，一般不会设定启动模式，否则在开机以后就会直接关机或重启，而无法正常使用系统。

表1-1 Linux系统的启动模式

启动模式 (run_level)	作用介绍
0	关机
1	单用户模式，只允许root账号登录
2	无NFS网络功能登录
3	文本模式登录
4	用户自定义模式登录
5	图形界面方式登录
6	重启

在确定启动模式后，进而执行/etc/rc.d/rc.sysinit程序进行操作系统的初始化，对系统的初始化主要是对系统进行初始化的设定，如设定主机名、系统时间、键盘布局等。然后进入目录/etc/rc.d执行run_level所表示

的目录rc.5，依次执行该目录中的所有内容。最终创建tty1~tty6共6个虚拟控制台，

注意

对于Ubuntu系统和Red Hat系统来说，启动文件是不同的，/etc/inittab是Red Hat系统中的文件，该文件中存储着和系统启动相关的文件。

1.2 Linux文件系统基础介绍

文件系统是操作系统对系统中所有文件存储空间的组织和分配，是操作系统基本的功能之一。文件系统主要负责文件的存储并对存入的文件进行保护和检索。具体地说，文件系统实现了对文件的增加、修改、删除以及检索的功能，在本节中将对Linux系统的文件系统结构进行讲述。

注意

在本书中所有的操作都是基于Ubuntu 10.04这一操作系统版本，而其他的操作系统和该版本存在一定的差别，但是其基本的使用方式一致。因此在后面的章节中将使用Ubuntu代替Linux系统。

1.2.1 必须了解的节点inode

在任何一个操作系统中都存在着数以千百计的文件，文件系统的作用就是对这些数量众多的文件进行组织和管理，从而及时响应用户的访问，并且使得文件不会丢失。常见的文件系统类型包括：

- fat系列
- ntfs
- ext系列

对于Linux系统来说，采用了与Windows系统完全不同的文件系统结构，在Windows系统中，常用的文件系统类型为NTFS或FAT32这两种形式，而在Linux系统中采用的是ext文件系统。ext文件系统是一种日志式文件系统，使用该类型的文件系统除了保存文件的内容之外，还存储了文件的许多属性和权限，因此，在Linux系统中，一般使用节点(inode)的方式来存储文件。

对于任何系统来说，系统中的文件都存储在磁盘中。磁盘中最小的存储单位是扇区，每一个扇区只有512个字节。对于现在的磁盘来说，扇区的数量是非常多的，并且不利于文件内容的存储。因此在Linux系统中实际上是按照块来进行存储的。一个块一般由8个连续的扇区组成。在块中，存储着文件的创建者、创建日期、上一次的修改日期、文件的大小等基本信息，这些储存文件基本信息的区域就称为inode，即索引节点，一个索引节点对应着一个文件，索引节点的内容如下：

- 文件的大小。
- 文件创建者，使用ID表示。

- 文件的所属组。
- 文件的读、写、执行权限。
- 文件的时间属性，包括上一次inode被修改时间、文件内容上一次修改时间、上一次打开时间。
- 链接数，包括软连接和硬链接两种。
- 文件中数据block的位置。

从上面的介绍中可以看出，在Linux系统中，一个完整的文件信息包含以下两部分：

- inode信息。
- 数据信息。

数据信息显然是存储文件的具体数据，而inode信息则存储文件的索引信息，该部分信息同样占用硬盘空间。每个inode节点的大小一般是128字节或256字节。inode节点的总数是在磁盘格式化时指定的。如果指定的inode节点全部被使用了，那么就不允许再创建新的文件。

注意

使用stat命令可以查看文件的索引节点inode信息，并且在索引节点中不包含文件名。

1.2.2 Linux系统文件结构

Linux系统的文件系统结构采用了一种称为树形结构的组合方式，

所谓树型结构就是所有的文件的父级目录最终都能归结到一个相同的目录中，这个目录被称为系统的根目录，使用符号“/”表示，在根目录中存在着其他的子目录。Linux系统的目录结构如图1-3所示。

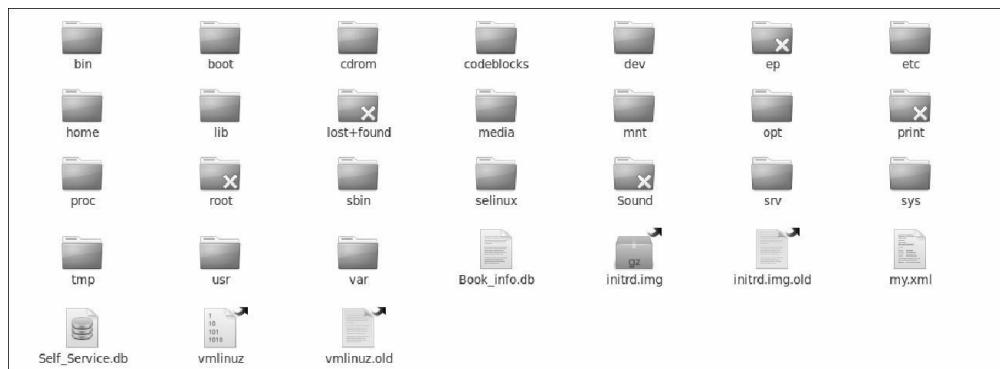


图1-3 Linux系统目录结构

从图1-3中可以看出，在根目录中包含很多目录，就像是树的枝叶一样，依赖于根的存在，从而“生枝发芽”。根目录中的每一个目录在Linux系统中都有着不可替代的作用。这些目录的作用如表1-2所示。

表1-2 Linux系统中目录及其作用

目录名称	作用
/	系统的根目录，包含系统中其他的所有文件
root	root用户的主目录，存放启动Linux系统的核心文件，如操作系统内核、引导程序Grub等
home	普通用户的主目录，存放普通用户的个人信息。按照系统中用户的登录名而使用不同的目录文件
bin	存放系统启动时需要执行的二进制文件
sbin	可执行文件目录，存放系统管理的命令，root用户或具有root权限的用户的才能执行里面的命令，普通用

	户无法使用
proc	存放Linux系统的所有内核参数以及系统的配置信息，按照进程的编号进行存取
usr	用户目录，存放用户文件
boot	存放系统启动时需要的文件
lib	存放系统需要的动态库以及核心模块
etc	存放各种配置文件
var	包含了运行时要改变的数据
mnt	用于挂载系统之外的文件系统。需要提前创建挂载的目录
tmp	临时文件目录，系统启动后的临时文件均存放 在/var/tmp中
lost_found	在文件系统修复时恢复的文件，系统错误时，存放错误的内容
opt	通常用来安装第三方的软件包

Linux系统按照表1-2中介绍的目录的作用对Linux系统中的文件进行处理，将所有的文件进行分门别类，按照每个文件的作用决定文件的存储位置。而对于用户来说，还有如下几个比较重要的目录：

- **usr** 目录
- **var** 目录
- **etc** 目录
- **proc** 目录

在上面4个目录中存储了Linux系统运行的常用信息，下面分别介绍这4个目录中存储的内容及其作用。

usr目录中存储的内容一般是用户文件，在usr目录中每个目录的作用也不尽相同。常用目录的作用如表1-3所示。

表1-3 usr目录中的内容及作用

目录	作用
bin	存放用户可以直接执行的所有的命令
sbin	存放与系统管理员相关的命令，如服务器的程序等，需要root用户或具有root权限的用户才可以执行
include	存放C和C ++语言的头文件，其他编程语言的相关文件在其他的目录中
local	本地安装程序的默认安装目录
man	手动生成的目录
info	信息文档
doc	安装包的文档信息

var目录一般包含了运行时要改变的数据，这些数据不通过网络与其他的计算机共享。常用的目录及其作用如表1-4所示。

表1-4 var目录中的内容及作用

目录	作用

local	/usr/tocal中安装程序的可变数据
lock	锁定文件，防止当前文件在使用时，被其他的程序修改
log	存储系统的各种日志文件，存储Linux系统的所有操作信息，如所有核心和系统程序信息
run	保存到下次引导前有效的关于系统的信息文件
spool	存储队列，涉及email、news、打印队列等
tmp	存储临时性文件，存储的文件比/tmp中的文件要大或存储的时间较长

在etc目录中，存放着系统中的所有的配置文件，如各种服务器、网络设备、系统的操作参数等文件，其主要的目录或文件如表1-5所示。

表1-5 etc目录中的内容及作用

目录	作用
rc或rc.d或rc*.d	存储系统的启动脚本或改变运行级别的脚本
passwd	系统的合法用户，包含用户名、主目录、密文形式的登录密码以及其他信息，按照一定的格式进行存储
group	存储用户组的相关信息
issue	登录提示符前的输出信息，通常包括系统的一段短说明或欢迎信息。内容由系统管理员确定
shadow	存储用户登录密码，明文和密文相对应，使用md5算法进行加密

profile	创建全局变量，一般存放的是环境变量
shells	包含可以使用的Shell

在proc目录中，存放着系统运行时的一些信息，如内存信息、设备信息、CPU信息等，这些信息记载着系统当前的运行状态，用户可以通过这些文件了解系统当前的运行状况，作为处理其他相关内容时的“参照”。Proc目录的内容如表1-6所示。

表1-6 proc目录中的内容及作用

目录	作用
1	存储进程init的信息，每一个进程号都有相应的目录文件存储相关信息
cpuinfo	存储CPU相关信息，如制造商、基本性能参数等
devices	存储当前运行的核心配置的设备驱动列表
dma	显示当前使用的DMA通道
loadavg	显示平均负载，指示系统当前的工作量
modules	显示当前系统加载了哪些核心模块
meminfo	存储物理内存和交换内存的实用信息
uptime	系统启动的时间
version	核心版本

上面介绍的几个目录是系统的主要目录，也是在实际应用中操作频

率较高的目录，但不表示其他的目录不重要，其他的目录中存储的信息在某些方面也是用户需要的内容，因此只有了解了所有的目录和文件的作用，就可以在需要的时候找到合适的目录，进而进行各种相应的处理。

注意

对于除了用户的主目录之外的目录，都需要root权限或相应的权限才可以读取里面的文件内容，否则会提示“权限不足”。

1.3 学会Linux系统的基本使用

在上面的章节中介绍了Linux系统的基本内容，如Linux系统的发展历史以及与Windows系统的区别，在本节中将详细介绍关于Linux系统的基本使用，如系统的登录与退出、Linux系统的启动过程、基本的系统选项配置等，这些内容是使用Linux系统的基本操作，因此需要读者熟练掌握。

1.3.1 系统的登录与退出

Linux系统在使用之前必须要成功登录才能进入系统，而在登录时需要输入用户名和密码。用户名和密码是用户在安装系统时设定的，除此之外还可以选择使用任何合法的用户名登录。所谓的合法用户名是在Linux系统中能够找到并且具有一定权限的用户。登录界面如图1-4所示。

示。



图1-4 Ubuntu 10.04系统登录界面

在图1-4所表示的登录界面中，可以使用默认的用户登录，就是显示出来的第一个用户，也可以选择其他用户登录。在登录时不管使用哪个用户登录，都需要输入用户的登录密码。默认用户的登录密码是在安装Ubuntu系统时设定的密码，而其他的用户密码可以通过root用户来进行设定或密码的修改。

输入密码后系统验证密码以及用户的合法性，如果用户名和密码匹

配成功，那么就会成功登录系统，如果不匹配，那么会提示用户名或密码错误，从而阻止用户的登录，如图1-5所示。



图1-5 输入用户登录密码错误

在输入用户的登录密码错误以后，会自动回到登录界面，允许再次输入用户名和密码进行登录。使用默认用户登录以后执行命令时只有普通用户的权限，除了使用默认的用户登录系统之外，还可以使用系统中存在的其他合法的用户登录系统，可以使用root用户登录或者使用其他的用户登录。如果需将用户的权限提高，可以使用root用户登录，从而将用户的操作权限提到最高，从而便于对系统进行操作。但是使用root用户登录以后，容易对系统造成误操作，从而对系统造成破坏。使用其

其他用户登录时，也需要输入用户名以及对应的密码才能登录，如图1-6所示。

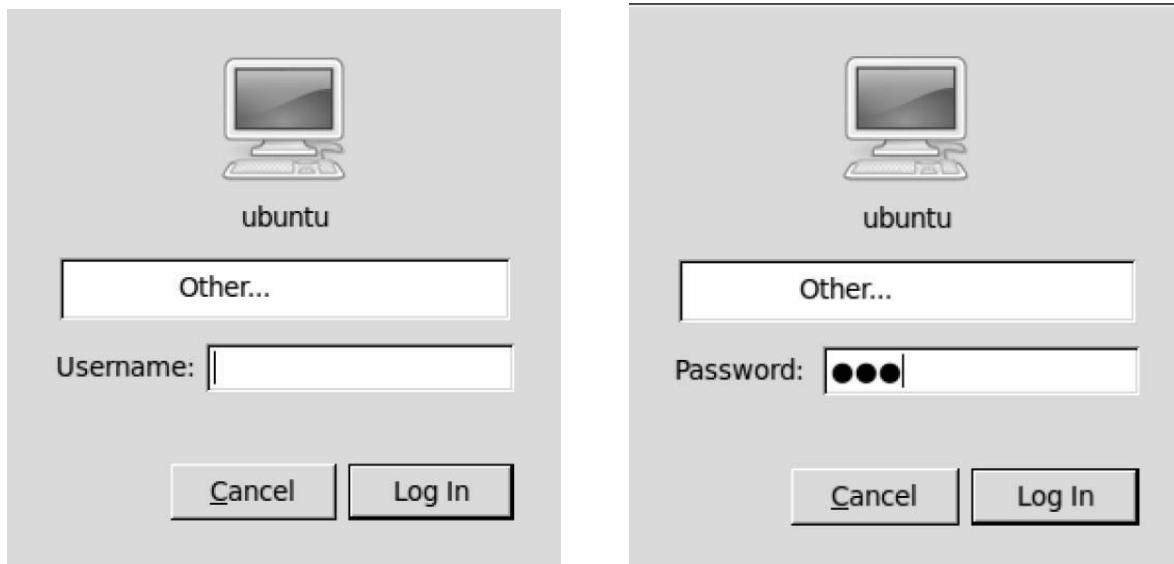


图1-6 使用其他用户登录并输入密码

注意

用户的密码可以通过命令进行修改，并且可以将用于登录的用户存储在特定的文件中，可以通过查看文件内容来明确可以使用哪些用户登录。

除了使用图形界面方式登录系统之外，还可以通过命令行的方式登录。在Linux系统中设置了6个虚拟控制台，每个控制台都可以单独访问Linux系统，进入这6个控制台的方式是使用组合键Ctrl+Alt+F1到Ctrl+Alt+F6，当按下相应的按键后，系统自动进入虚拟控制台，使用合法的用户名和密码登录后，就可以操纵Linux系统了。

当系统使用完毕之后，需要退出系统。对于一般的Linux系统来说，可以通过以下3种方式退出：

- 单击图标
- 使用命令init
- 使用命令shutdown

对于退出Linux系统来说，比较简单的方式是使用图形化操作环境，可以直接单击面板上的【关机】按钮退出系统。单击【关机】按钮以后，显示内容如图1-7所示。

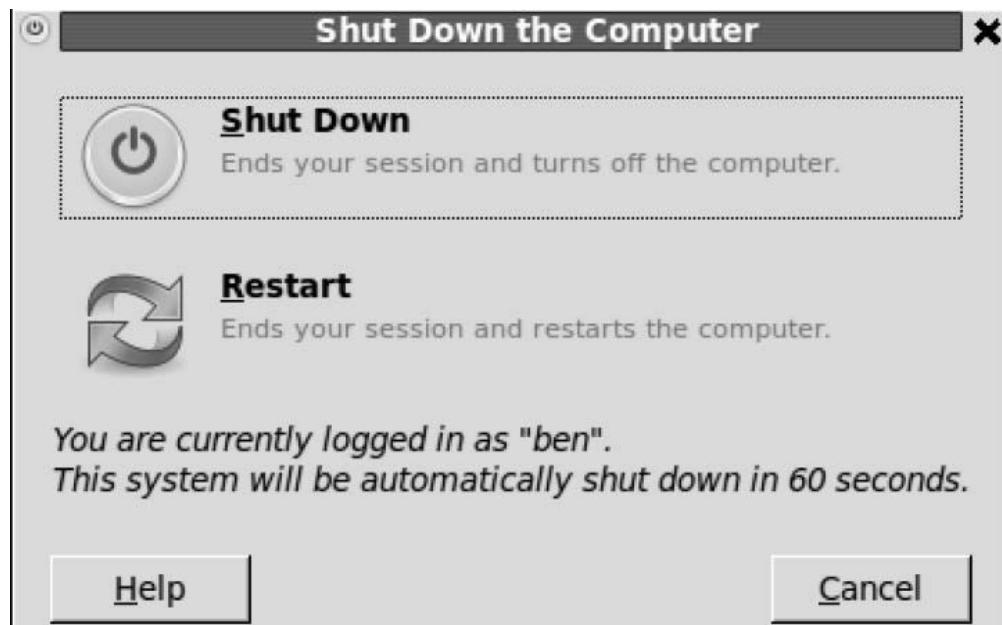


图1-7 关机界面

在弹出关机界面之后，可以根据需要选择重启系统还是直接退出系统，也可以单击【Cancel】按钮退出关机操作。

1.3.2 系统基本选项配置

在成功登录Linux系统以后，可以根据用户的实际需要对系统进行各种设置。在安装的过程中，也需要进行一些必要的设置。对于Linux系统来说，可以在【系统首选项】菜单中进行配置，【系统首选项】的部分内容如图1-8所示。



图1-8 【系统首选项】菜单

图1-8中显示了【系统首选项】菜单的部分内容，在该菜单中，可以实现网络连接的配置、鼠标的各种属性的配置以及其他常用设备的配置方式。读者可以在实际操作时，将每一个界面点开，查看具体的操作。

注意

【系统首选项】菜单的顶部和底部都各有一个箭头，说明该菜单只显示了部分内容，在上面和下面还有部分内容未显示。

1.3.3 文本编辑器的使用

在Linux系统中常用的文本编辑器为Gedit、vi（vim）系列、Emacs等。Gedit编辑器类似于Windows系统中的记事本，该编辑器在图形界面系统中才能够使用，而vi（vim）以及Emacs编辑器属于文本编辑器，这两类编辑器即使没有图形界面也能够正常使用，虽然它们没有华丽的界面，也不能使用鼠标辅助操作，但是当图形桌面系统“挂掉”以后，文本编辑器就成了唯一的选择，因此学会使用文本编辑器是操作Linux系统的基础技能之一。

vim是vi的增强版，而vi是Linux系统中使用较早的文本编辑器，一般的Linux系统中都默认安装了vi编辑器。下面就介绍vi编辑器的使用方式。

1. 启动vi

vi属于文本编辑器，没有图形化的编辑环境，因此在Shell终端中输入vi命令后就能打开vi编辑器，从而进行文本的编辑。在启动vi编辑器时，一般还会使用一些选项来对打开的文件进行某些特定操作。常用的打开vi编辑器的方式如表1-7所示。

表1-7 vi编辑器打开方式

vi命令常用选项	功能
vi	打开vi文本编辑器
vi filename	打开文本编辑器，并且文件的名字为filename
vi -R filename	只读方式打开文件名为filename的文件

对于表1-7中的打开vi编辑器的方式来说，直接使用vi命令将只能打开vi编辑器，在退出时需要保存文件名。除此之外还可以在打开编辑器时指定文件名，或者使用-R选项以只读的方式打开文件。对于一般情况来说，在打开vi编辑器的同时指定文件名，其操作方式如下所示。

```
ben@ben-laptop:~$ vi test.txt
```

在执行了上面的命令之后，vi编辑器就会根据文件名自动创建相应的文档，并且打开编辑器，从而使得用户可以对文件进行编辑。vi文本编辑器打开后的界面如图1-9所示。

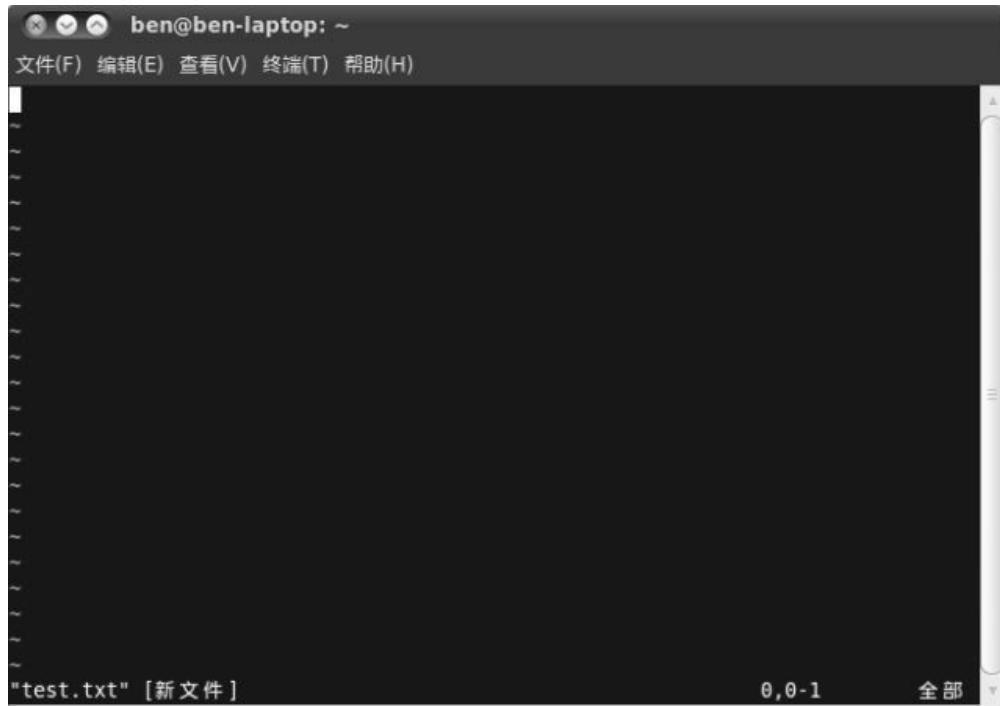


图1-9 vi文本编辑器界面

刚打开的vi编辑器默认处于命令行状态，此时是不能进行任何的文本输入的，只有转为插入模式才能进行文本的编辑。

注意

如果打开的文件不是空文件，就会在vi编辑器中显示文件的内容。

命令行状态也可称为底行状态，因为所有的操作都是在vi编辑器的底部完成。在命令行模式下可以对文件进行各种操作，此时所有输入的内容都被当做命令来处理，常用的命令可以分为如下几类：

- 文本删除类

- 文本插入类
- 屏幕翻滚类
- 光标移动类

常用的文本删除类命令及其作用如表1-8所示。

表1-8 底行模式中常用的文本删除类命令

命令名称	作用
x	删除光标所在字符
dd	删除光标所在行中所有的字符
r	修改光标的位置
R	进入字符替换，使用新增文字替换原来的文字，可以按ESC键退出字符的替换
s	删除光标所在的字符，并进入编辑模式
S	删除光标所在的行，并进入编辑模式

在底行模式中可以使用相关的命令实现文本的删除操作，可以删除单个字符，也可以删除文本中的整行内容，并且能够进行简单的文字替换功能。

在vi编辑器中不能使用鼠标进行辅助操作，因此，只能使用各种命令来进行光标的移动，常用的光标操作方式如表1-9所示。

表1-9 光标操作常用命令

命令名称	作用
h	光标左移一个字符
l	光标右移一个字符
space (空格键)	光标右移一个字符
BackSpace (字符删除键)	光标左移一个字符
k或Ctrl+p	光标上移一行
j或Ctrl+n	光标下移一行
) (右小括号)	光标左移至行首
((左小括号)	光标左移至行尾
H	光标移至屏幕顶行
M	光标移至屏幕中间行
L	光标移至屏幕最后行
0	光标移至当前行首
\$	光标移至当前行尾

如果打开的文件内容过多，会造成一屏幕无法全部显示，那就需要进行屏幕的滚翻，在vi编辑器中，屏幕滚翻相关的命令如表1-10所示。

表1-10 常用屏幕滚翻类命令

命令名称	作用
Ctrl+u	向文件首翻半屏
Ctrl+d	向文件尾翻半屏
Ctrl+f	向文件首翻一屏
Ctrl+b	向文件首翻一屏
nz	将第n行滚至屏幕顶部，不指定n时将当前行滚至屏幕顶部。

vi编辑器的主要功能就是对文本文件进行编辑，在编辑时，可以在文档的各个位置进行数据的插入。常用的数据插入命令如表1-11所示。

表1-11 常用数据插入命令

命令名称	作用
i	在光标前插入
I	在当前行的行首插入
a	在光标后插入
A	在当前行的行尾插入
o	在当前行之下新开一行
O	在当前行之上新开一行
r	替换当前字符
R	替换当前字符及其后字符，直至按ESC键

s	当光标的当前位置开始，以输入的文本替换指定个数的字符
S	删除指定数目的行，并以输入的文本替换删除的内容

通过上面的表可以看出，熟练使用vi编辑器需要掌握很多命令。虽然命令相对较多，但是每一个命令在进行文本的编辑时都可能用到，因此需要读者花大力气进行反复的练习，从而掌握这些基本命令的使用。

上面的4类命令是使用vi编辑器常用的命令，除此之外还有其他许多命令需要读者了解，在此不做赘述，有兴趣的读者可以通过阅读其他相关资料进行学习。

2. 文件编辑

在打开编辑器以后就可以根据实际需要进行文本的编辑了。但是刚打开的vi编辑器处于命令行模式，只有在输入特殊的命令字符后才可以进行文本的编辑操作。vi编辑器的这种操作模式称为命令行模式，除了命令行模式以外，vi编辑器还存在插入模式、底行模式。在进行文本编辑的过程中，这3种模式相互转化，从而完成整个文本的编辑过程。在不同的模式中需要使用不同的命令对编辑器进行各种操作。

输入字符i可以将vi编辑器从刚开始的命令行模式变成插入模式，这样就可以通过键盘输入字符，进行文本的编辑。编辑完成后如图1-10所示。

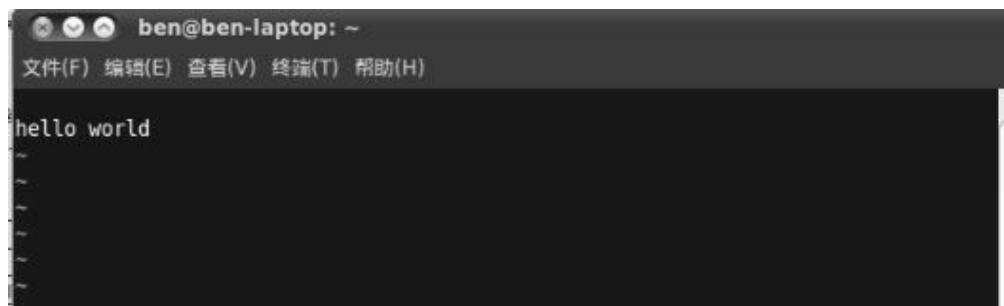


图1-10 vi编辑器的插入模式

在进行文本的编辑时不可以使用鼠标进行辅助操作，但是可以使用一些命令代替鼠标操作，常用的命令如表1-12所示。

表1-12 vi编辑器插入模式中的常用命令

命令名称	作用
i	在光标前
I	在当前行首
a	光标后
A	在当前行尾
o	在当前行之下新开一行
O	在当前行之上新开一行
r	替换当前字符
R	替换当前字符及其后的字符，直至按ESC键
s	从当前光标位置处开始，以输入的文本替代指定数目的字符
S	删除指定数目的行，并以所输入文本代替之

当编辑文本结束后，需要及时保存文件并且安全退出vi编辑器，否则刚编写的文本内容不会储存在内存中，再次打开后也不会显示在文件中。在插入模式中按下键盘的ESC键，这样就可以从插入模式进入底行命令模式。在进入底行模式后，需要先输入冒号“：“，然后再输入相应的命令，这样命令才能够执行。在底行模式中常用的命令如表1-13所示。

表1-13 底行模式中的常用命令及其作用

命令名称	作用
q	退出vi编辑器
!	强制执行某个命令
x	保存文件并且退出vi编辑器
E	创建新的文件名，并可以为文件命名
N	在当前vi编辑器窗口打开新的文件
w	保存文件，但是不退出vi编辑器
set nu	显示行号
/	查找匹配字符串功能
?	查找特定字符串
%	表示所有行

在底行模式中，常用的命令为q和w命令，这两个命令还可以和强

制退出符号“!”一起使用，从而使得保存文本后，强制退出vi编辑器，但是一般不建议使用强制退出符号。退出vi编辑器的操作方式如图1-11所示。

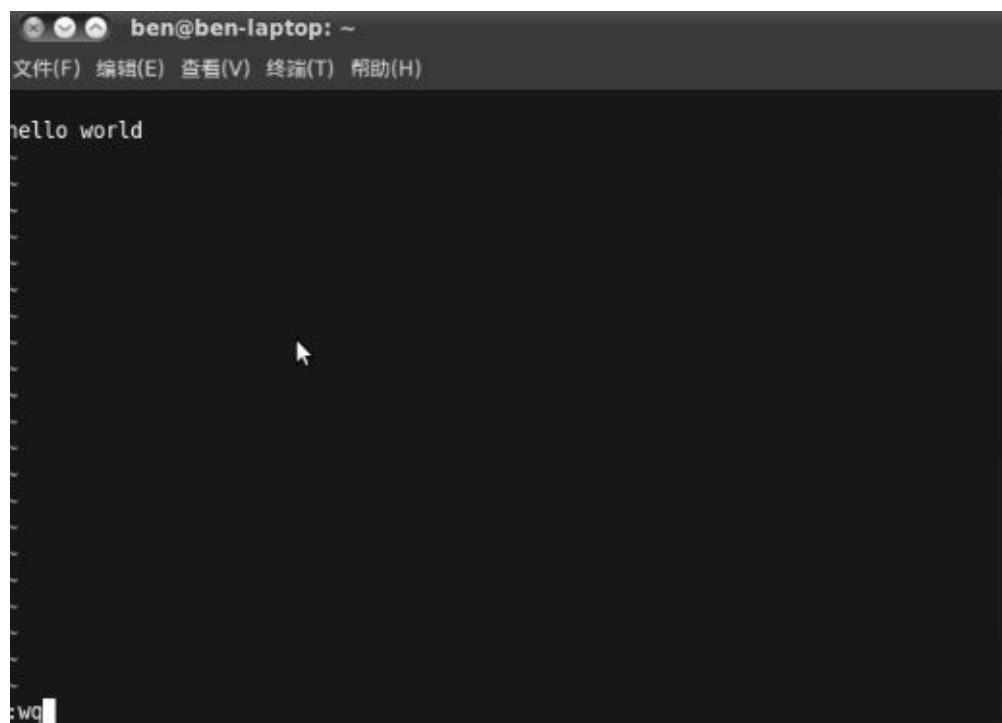


图1-11 退出vi编辑器

在退出vi编辑器之后，文档的编写就完成了，可以通过其他命令（如cat命令）来查看编辑的内容是否正确，如果不正确，还可以进行修改。也可以使用ls命令来查看新文件是否已经存在，如下所示：

```
ben@ben-laptop:~$ ls -l test.txt
-rw-r--r-- 1 ben ben 12 2014-05-29 21:25 test.txt
ben@ben-laptop:~$ cat test.txt
hello world
```

使用vi编辑器时要熟练掌握3种模式的相互转换方式以及能够进行

的操作。在命令模式中可以通过命令来完成光标的定位、字符串的检索、文本的恢复/修改/替换/标记、行结合及文本位移等功能；插入模式能够实现文本的编辑功能；底行命令模式主要完成文本的全局替换、在文本中插入Shell命令、vi编辑器的设置、文本的存盘退出、文本块的复制、多个文本间的转换及缓冲区的操作功能。这3种模式的相互转换关系如图1-12所示。

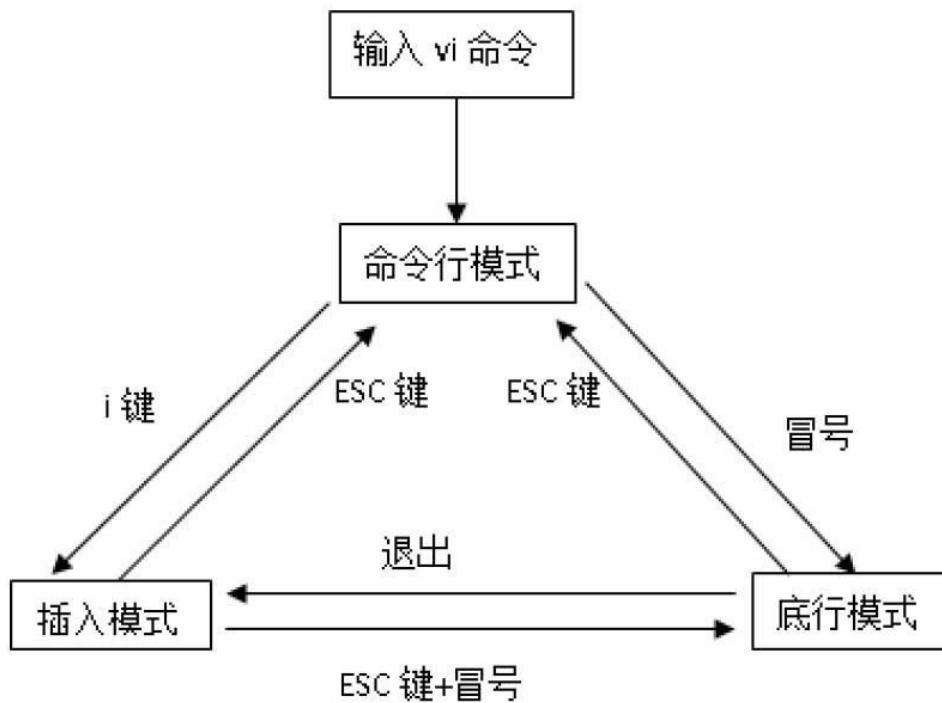


图1-12 vi编辑器3种模式相互转换示意图

通过图1-12可以看出，输入vi命令之后，默认进入命令行模式，而从命令行模式到插入模式只需要键入i键。当编辑完成后，可以使用ESC键+冒号的组合键进入底行模式，从而退出vi编辑器。从底行模式到命令行模式以及从插入模式到命令行模式，只需要按ESC键就可以实现。

注意

`vim`编辑器的使用方式和`vi`相同，唯一的不同是创建文件时需要使用`vim`命令，而不是`vi`命令。

1.4 走进Shell

X Windows等图形化系统的使用，使得Linux系统的操作变得和Windows系统一样方便，不少操作使用鼠标就可以完成。然而，许多的Linux系统功能必须使用Shell命令才能更加方便，并且效率要比使用图形化界面更高。虽然Shell没有图形界面那么直观，但是Shell伴随着Unix/Linux的发展，其功能也日趋完善。而且，在大部分的Linux服务器中是没有图形界面的，有的只有Shell。

1.4.1 什么是Shell

Shell的中文意思为“贝壳”，在Linux系统中，Shell就类似于Linux系统的“贝壳”，在Linux系统的外部起着保护系统的作用，同时完成用户与Linux系统之间的数据以及信息的交互。对于任何系统来说，内核都是非常重要和脆弱的，如果内核受到损伤，会直接影响系统的稳定性。Shell的出现就起到了保护内核的作用。因为有了Shell以后，用户不会直接和内核进行“交流”，而是通过Shell进行信息和数据的转发，从而避免了内核直接暴露在用户面前，防止因为用户误操作造成内核的“受伤”。

Shell实际上是一个命令解释器，将用户的请求进行处理，翻译成Linux系统内核能够处理的内容。当需要执行的命令执行结束以后，将

执行结果翻译成用户能够看懂的信息，并且显示在屏幕上。

Shell还是一种编程语言，其源文件一般被称为Shell脚本。Shell脚本类似于Windows系统中的批处理文件，能够将需要执行的命令放到脚本中，并且在脚本中能够定义变量、使用各种控制结构对命令的执行顺序做出限制。

1.4.2 Shell在Linux系统中的作用

Shell在Linux系统中的作用如图1-13所示。

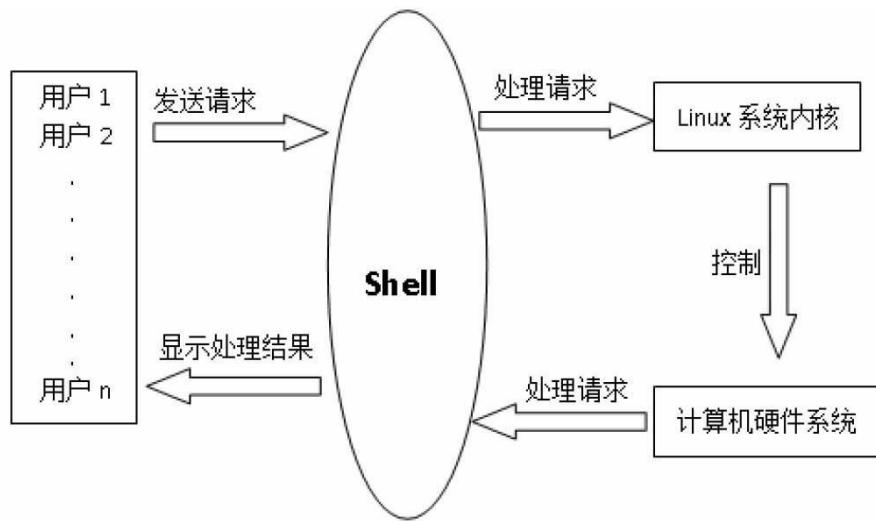


图1-13 Shell在Linux系统中的作用示意图

在图1-13中可以看出，Shell在Linux系统中的作用非常重要。当任何用户需要计算机完成某项功能时，发送的请求以Shell命令的方式首先由Shell解释成Linux内核能够理解的内容，然后才由Linux系统内核通过各种控制方式由特定的硬件设备完成用户的请求。而用户请求的最终结果则首先发送给Shell，然后由Shell解释成用户能够理解的方式并在适当

的地方显示出来。

在使用Linux系统时，Shell具有双重含义。一种含义是指代Shell解释器，用来将用户输入的命令解释成内核能够“看懂”的语言，并且把命令的执行结果解释成用户能够看懂的信息。另外一种含义就是Shell脚本，可以理解为一种程序设计语言，在Shell脚本中定义了各种变量、各种内置参数，还能使用循环结构、分支结构等复杂的程序控制结构，可以用来实现“自动化”的处理。

1.4.3 Shell的种类

随着Linux系统的发展，Linux系统中存在多种Shell可以使用，常用的Shell的类型如下：

- Bourne Shell
- Bash Shell
- C Shell
- Korn Shell

Bourne Shell是首个可以使用的Shell版本，在1977年底引入。Bourne Shell 既是一个交换式的命令解释器，又是一种命令编程语言。而对于一般的Linux系统来说，默认的Shell类型是Bash Shell。该类型的Shell源自于Bourne Shell（即sh）。Bash的全名为Bounce again Shell，是Linux系统发展早期较为重要的Shell版本，是GNU计划的一部分，用来替换Bourne Shell。当登录时，Shell会读取/etc/profile文件和用户主目录中的.profile文件，从而加载一些特定的信息，如环境变量等。

C Shell的主要作用是让用户更容易使用交互式功能，并且把ALGOL风格的语法变成了C语言的风格，从而方便了C程序员对Shell的操作。相对于Bourne Shell来说，C Shell新增了命令历史、别名、文件名替换以及作业控制等新功能。

在Linux系统发展的初期，只有两种Shell可以使用，一种是Bourne Shell，另外一种是C Shell，为了改变这种局面，Korn Shell被开发出来。Korn Shell融合了Bourne Shell和C Shell的优点，结合了C Shell的交互性，融合了Bourne Shell的语法，并且增加了数学计算、行内编辑等功能，

对于不同的操作系统来说，默认使用的Shell类型也不相同。不同系统的默认Shell如表1-14所示。

表1-14 不同系统的默认Shell

系统名称	默认 Shell 类型
AIX	Korn Shell
Solaris	Bourne Shell
FreeBSD	C Shell
HP_UX	POSIX Shell
Linux	Bash

表1-14展示了不同系统中的默认Shell类型。本书主要以Bash作为默认的Shell类型，其他类型的Shell脚本的编写方式略有不同，在本书中暂不做介绍。有兴趣的读者可以参照相关的内容进行学习。

注意

通过环境变量\$SHELL可以判定当前系统默认的Shell类型。

1.5 小结

本章是介绍Shell脚本的第一章，也是奠定Shell脚本基础的一章。本章主要介绍了Linux系统的一些基础内容，包括Linux系统的发展、Linux文件系统的结构、Linux系统的基本使用以及Shell的基础介绍。

Linux系统是一种类Unix系统，是林纳斯·托瓦兹出于个人兴趣爱好而开发出来的新的系统。该系统是一种free系统，它不单单是免费的系统，还可以自由地修改和发布，并且与Windows系统存在着本质的区别。

Linux系统的文件系统是采用日志式文件系统ext格式，目录结构采用树形结构，每一个目录都存储着和系统相关的一部分内容。在系统中文件按照索引节点inode来进行存储，索引节点中包含着除文件内容和文件名之外的所有文件的基本信息。

Linux系统自使用之前需要登录，用户可以使用默认用户登录，还可以使用root用户登录。登录时需要输入用户对应的密码。只有用户名和密码相匹配，才可以进入系统。进入系统之后，可以通过菜单【系统管理】和【系统首选项】来对系统进行一些个性化设置。

Shell在Linux系统中既是一个“保护壳”，也是一个命令解释器，同

时还是一种编程语言。Shell作为Linux内核和用户的沟通桥梁，可以将用户的需求转换成内核能够执行的内容，同时还可以将内核运行的结果转换为用户可以看懂的内容。在不同的系统中，使用的Shell版本也不同，可以通过环境变量来查看默认的Shell是哪一种。

第2章 迈出Shell脚本编程第一步

在Windows系统中经常需要使用批处理文件，就是使用一些DOS命令批量实现特定的功能。在Linux系统中，也存在这样的一类“批处理”文件。在文件中填写需要执行的Shell命令，使用这些Shell命令来解决一些特定的问题，这些“批处理文件”就是Shell脚本。本章作为Shell脚本的基础章节，主要介绍关于Shell脚本的基础知识，在后面的章节中将逐步介绍如何编写Shell脚本。

本章的主要内容如下：

- Shell脚本语言概述，包括Shell脚本基础内容以及如何使用Shell终端
- Shell命令格式
- 如何编写最简单的Shell脚本

2.1 Shell脚本语言概述

脚本语言是一种是特定的描述性语言，按照特定的格式编写成可执行文件，然后由计算机进行执行。一般来说，每一种系统中都会存在一种或多种脚本语言，来供用户使用这些语言编写相应的脚本程序，从而完成一些高级语言不适合完成的任务。

在Linux系统中，常用的脚本语言包括Shell脚本语言和Python脚本语言，本书中将主要介绍Shell脚本语言。对于Python脚本语言的使用，有兴趣的读者可以参考其他相关书籍来进行学习。

2.1.1 Shell脚本语言的定义

Shell脚本，类似于Windows系统中的批处理命令，将Shell命令放入一个文本文件中，用正规表达式，实现管道命令、重定向等功能，还可以使用分支、循环以及逻辑判断等控制结构，以达到我们所想要的处理目的。通过执行该文本文件，执行放入文件中的所有的命令。这样可以方便系统管理员对系统进行批量化设置和处理，使用Shell命令来完成基本的操作，使得其执行效率比其他编程语言更高。

Shell脚本语言简单、易学、易用，而且还有很多是在Shell脚本中使用的小工具，如正则表达式、sed、gawk等，Shell脚本特别适合处理文件和目录之类的对象，从而以最简单的方式快速完成某些复杂的事情。

Shell 脚本语言是Linux/Unix系统上一种重要的脚本语言，在Linux/Unix领域应用极为广泛，熟练掌握Shell脚本语言是一个优秀的Linux/Unix开发者和系统管理员的成长必经之路。利用Shell 脚本语言可以简单地实现复杂的操作，而且Shell 脚本程序往往可以在不同版本的Linux/Unix系统上通用。

2.1.2 Shell终端的基本使用

Shell命令需要在Shell终端中才能运行。Shell终端实现了命令解释器的作用，在安装完系统以后，一般都会默认安装Shell终端。打开Shell终端时需要按照下列路径进行：【应用程序】→【附件】→【终端】。打开Shell终端后，其窗口如图2-1所示。

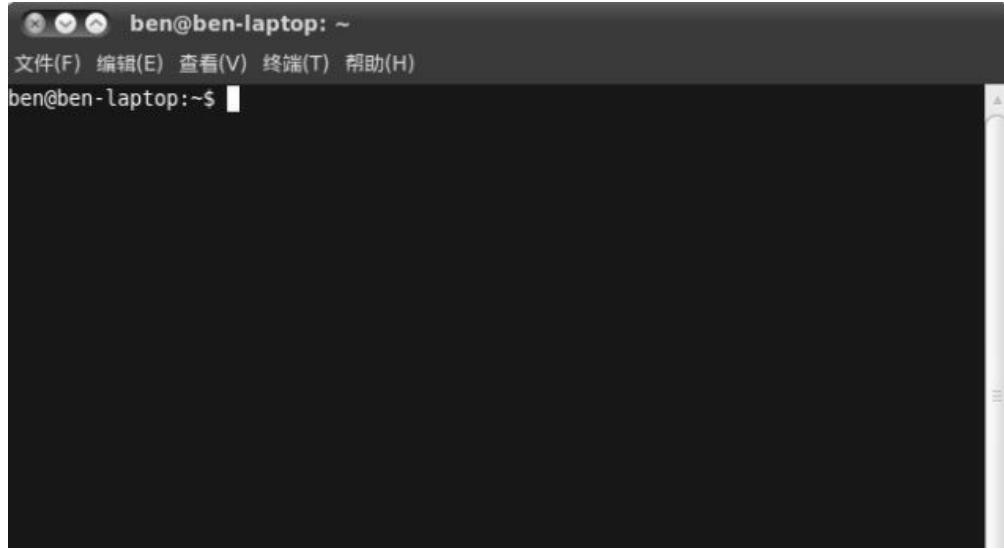


图2-1 打开的Shell终端示意图

在Shell终端中，输入的内容将从光标处开始，当命令输入完成之后，需要按下Enter键，那么Shell就将自动解释命令，并执行命令。

Shell终端的窗口和普通的Linux应用程序窗口一样，左上面的3个按钮分别是【关闭】按钮、【最小化】按钮、【最大化】按钮，这3个按钮对应着窗口的关闭、最小化到任务栏、全屏幕显示的功能，而在最上面显示的内容是Shell的命令提示符，显示了Shell终端当前的信息。

Shell终端的菜单栏中提供了Shell终端的常用功能，可以对Shell终端进行简单的设置。常用的设置包括设置Shell终端的首选项以及查看编码方式等。下面将对这些主要菜单的使用方式进行逐一介绍。

2.1.3 Shell终端菜单的使用

Shell终端和普通的窗口一样，也存在菜单栏，在菜单栏中也有一些比较重要的菜单项。对于Shell终端来说，其菜单栏如图2-2所示。



图2-2 Shell终端菜单栏

从图2-2中可以看到，Shell终端的菜单栏中都是些简单并且常用的菜单项。其中比较常用的菜单项有用于对Shell终端的基本配置进行修改的菜单【编辑（E）】中的【编辑配置文件】菜单项和用于打开子Shell标签的【文件（F）】中的【编辑配置文件】菜单项。该菜单项可以对Shell终端的基本配置进行修改，打开该子菜单以后，其内容显示如图2-3所示。



图2-3 【编辑配置文件】菜单项内容示意图

通过图2-3可以看出，在子菜单中可以对Shell终端进行一些设置，如对标题、颜色、背景色等进行相关的设置。在进行了相应的设置以

后，Shell终端会立即按照最新的配置进行显示，除非再改回原来的配置。

2.2 Shell命令格式介绍

Shell脚本的基本元素是Shell命令，在Shell中执行任何命令都需要注意并不是每个命令的使用方式都是相同的，每个命令有每个命令特定的要求，因此需要注意Shell命令的格式。

2.2.1 Shell命令格式

在每一次输入命令之前，在输入光标的前面都会出现同样的文字，这些文字在输入语句的前面始终存在，但是会随着系统中一些信息的变化而发生改变，这些信息的作用如图2-4所示。

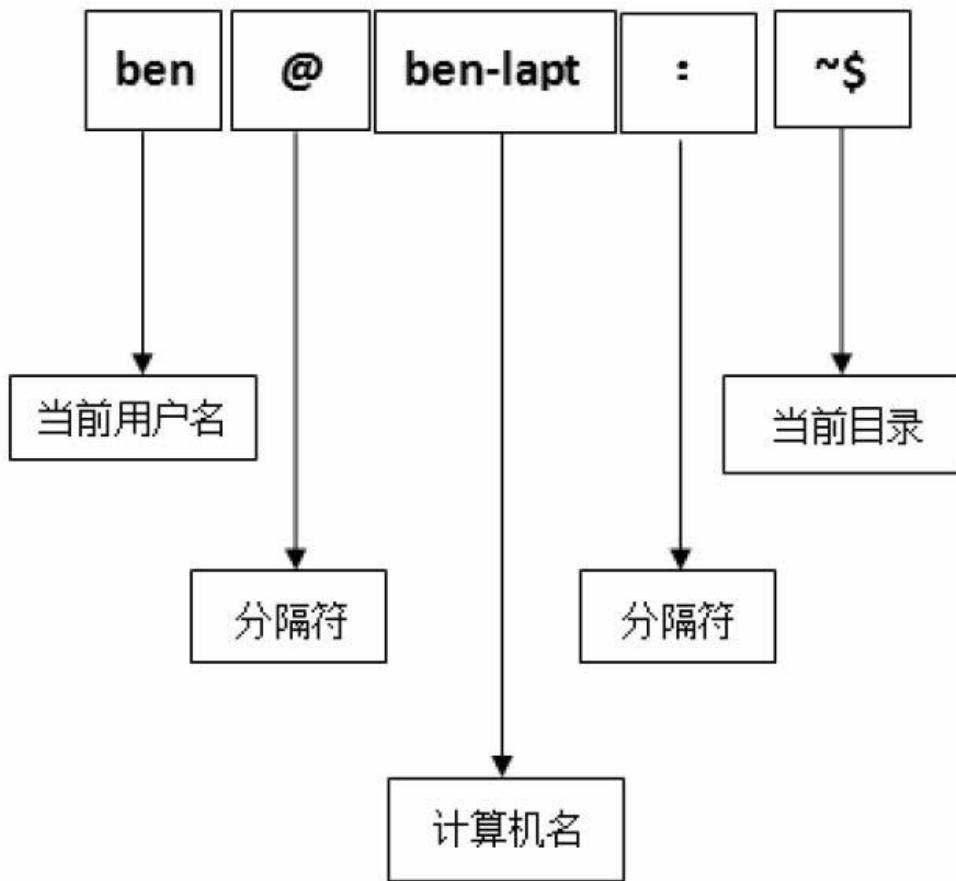


图2-4 命令提示符中符号的作用

通过图2-4可以看出，这些内容最前面的部分是当前用户的用户名。在图2-4显示的命令行提示符中，可以看到使用的用户名是ben用户，在符号“@”的后面是计算机名，这些在安装系统的时候会有专门的界面提示输入。命令行提示符会随着目录的变化而发生变化。当前登录用户发生变化时，符号“\$”也会变成其他的字符。在Linux系统中，一般只有如下两种用户：

- 普通用户
- 超级用户

普通用户使用的提示符是美元符号“\$”，而超级用户使用的符号是

波浪线“～”，因此可以通过Shell终端中的命令提示符来判定当前的用户是普通用户还是超级用户，从而进一步得出是否具有操作某些命令的权限。

2.2.2 命令行界面介绍

命令行界面就是在提示符下输入命令的界面，如打开的Shell终端界面以及在Windows系统中使用的DOS界面等。在命令行界面中没有图形化的操作环境，只能使用系统能够执行的命令完成一定的操作。虽然其操作不是很方便，但是其执行速度要比图形化操作更快，功能也更加强大。不足之处就是需要了解系统中存在哪些可以使用的命令，并需要了解每个命令的使用方式以及作用。

传统的Unix环境就是 CLI（Command Line Interface，命令行界面），包括最终的Linux系统也是使用命令行界面。在命令行界面中，只能在命令行下键入相应的命令来执行想要的操作。对于初学者来说，没有了鼠标的辅助操作后，只能依靠于键盘的操作，这样的操作显然是非常复杂的，但是其操作效率却比图形界面高，功能也更强大。

在Ubuntu系统中，除了存在桌面环境以外，还同时存在6个命令行界面，在命令行界面中，没有方便的鼠标操作和绚丽的图形化操作环境。在命令行环境中，只能使用系统支持的Shell命令来完成所有的操作，不能使用鼠标进行操作，类似于Windows操作系统中的DOS环境。在跳转到命令行环境时，首先需要使用正确的用户和密码进行登录，只有通过验证的用户才可以对系统进行操作。而在输入密码的时候，输入的内容是不可见的，因此要注意输入的正确性。登录部分如图2-5所

示。

```
Ubuntu 10.04 LTS ubuntu tty1
ubuntu login: ben
Password:
Last login: Sun Jul 13 19:41:57 PDT 2014 on tty1
Linux ubuntu 2.6.32-21-generic #32-Ubuntu SMP Fri Apr 16 08:10:02 UTC 2010 i686
GNU/Linux
Ubuntu 10.04 LTS

Welcome to Ubuntu!
 * Documentation: https://help.ubuntu.com/
 New release 'precise' available.
 Run 'do-release-upgrade' to upgrade to it.

ben@ubuntu:~$ _
```

图2-5 命令行界面的登录

通过图2-5可以看出登录成功后，就可以对系统进行操作了。在进行操作时，只能使用Shell命令来完成各种操作，而对于其他的，在图形化操作环境中可以使用的东西将不再使用。Ubuntu 10.04系统的命令行界面如图2-6所示。

```
ben@ubuntu:~$ ls
ben@ubuntu:~$ cd /
ben@ubuntu:/$ ls
bin dev initrd.img media proc selinux tmp vmlinuz
boot etc lib mnt root srv usr
cdrom home lost+found opt sbin sys var
ben@ubuntu:/$
```

图2-6 Ubuntu 10.04系统的命令行界面

通过图2-6中显示的内容可以看出，在登录系统的时候也需要使用输入登录用户的密码，只有输入正确密码后才能使用系统。登录成功后，直接进入命令行界面。命令行界面和DOS界面类似，都是只能使用Shell命令来完成操作。而不再有图形化的操作环境。因此，在命令行界面进行操作，需要详细了解Ubuntu系统和常用的Shell命令，否则，在没有图形界面的环境中，操作将会变得极为困难。

注意

Linux系统中的命令行环境一般被称为虚拟终端，可以使用组合键Ctrl+Alt+F1至Ctrl+Alt+F6进行切换，每一个命令行环境都是作为一个独立的系统存在，而跳转回桌面环境，可以使用组合键Ctrl+Alt+F7。

2.2.3 如何获取帮助

在Linux系统中，有一些命令类似于Windows系统中的F1键，能够通过它们获取相关的帮助信息，可以帮助使用者了解特定命令的使用方式或其他信息，熟练使用帮助命令可以使得初学者更快地学会Shell的使用以及Shell命令，从而为Shell脚本的编写打好基础。常用的帮助命令是man、info等，关于这些帮助命令的使用下面将一一进行介绍。

man命令可以获取某个命令的在线帮助手册，然后按照特定的格式进行输出。其使用方式一般为man命令+待查找命令，man命令的使用方法如下：

```
[app_usr@js4app24 settle]$ man man
```

在上面的示例中，实现了查找man命令的帮助手册，而显示的内容如下：

```
[app_usr@js4app24 settle]$ man man  
man (1)
```

NAME

man - format and display the on-line manual pages

SYNOPSIS

```
man [-acdfFhkKtwW] [--path] [-m system] [-p string]
      htmlpager] [-S section_list] [section] name ...
```

DESCRIPTION

man formats and displays the on-line manual pages. It reads the manual pages from the system's manual page database.

See below for a description of where man looks for the manual pages.

OPTIONS

-C config_file

Specify the configuration file to use; the default is /etc/man.config.

-M path

Specify the list of directories to search for manual pages. The default is /usr/share/man.

-P pager

Specify which pager to use. This option overrides the PAGER environment variable.

-B browser Specify which browser to use on HTML files. The default is lynx.

.....//省略部分帮助信息

-F or --preformat

```
Format only - do not display.
```

```
-h      Print a help message and exit.
```

```
:
```

上面的内容是整个man命令的在线帮助手册，NAME部分表示该手册是关于该命令的在线帮助手册；SYNOPSIS部分是该命令的大体用法，里面包括常用的选项以及选项组合；DESCRIPTION部分是对该命令的详细描述，在该部分内容中详细介绍了命令的使用以及在使用过程中的注意事项。通过该部分内容的介绍可以了解当前命令的使用方式；OPTIONS部分介绍了该命令可以使用的选项以及选项含义。

在帮助手册的最后面为一个冒号“：“，在冒号的后面可以输入相应的命令对帮助手册进行操作，而一般的操作就是使用回车键查看当前屏幕中未显示的部分，还可以使用【q】键退出帮助手册。冒号以外的部分是不允许进行任何操作的。

当遇到不熟悉的命令时，可以使用man命令查找在线帮助手册，从而详细了解该命令的使用方法。

在Shell中使用的任何命令都是事先存储在系统中的，如果使用的命令在系统中不存在，会出现如下错误提示：

```
ben@ben-laptop:~$no_exsit
bash: no_exsit:  not found
ben@ben-laptop:~$
```

当出现上述的错误时，说明使用的命令在系统中不存在。此时需要

通过安装中心或使用apt-get工具进行当前使用命令的安装，关于apt-get工具的使用方式，将在后面的章节中进行讲述。

2.3 第一个Shell程序：Hello, Bash Shell！

任何编程语言都会有一个入门程序，对于其他编程语言来说，一般使用hello world作为入门程序，而对于Shell脚本的入门程序来说，也采用相对简单的语句来介绍Shell脚本。

2.3.1 创建Shell脚本

Shell脚本的创建可以按照一定的流程进行操作，创建Shell脚本的大致流程如图2-7所示。

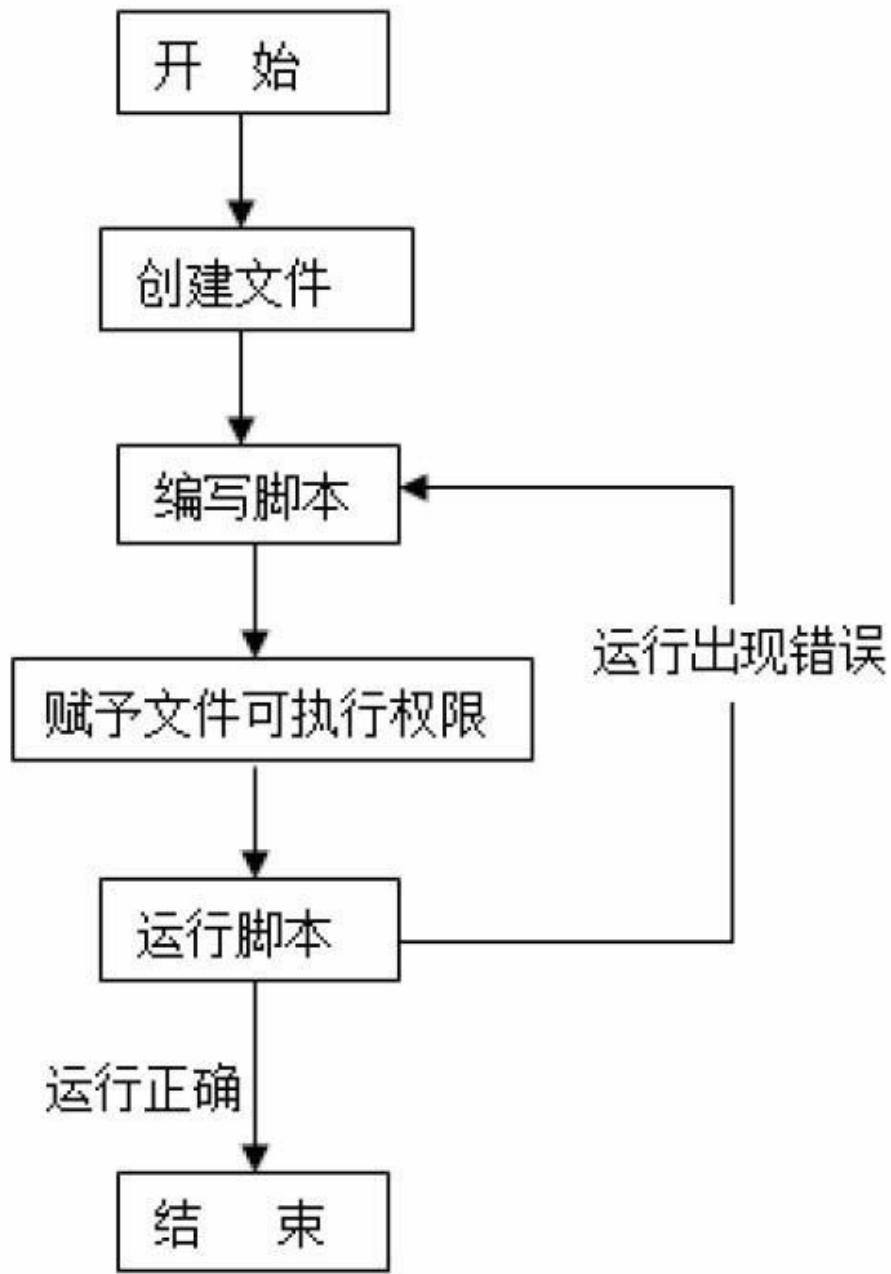


图2-7 Shell脚本创建过程

通过图2-7可以看出，在编写Shell脚本时，首先要创建脚本文件。脚本文件实际上为普通的文本文件，因此可以使用touch命令直接创建对应的脚本文件，也可以通过使用文本编辑器gedit、vi（vim）等新建的方式创建Shell脚本，甚至可以使用鼠标右键快捷方式创建脚本文件。

在此只介绍文本编辑器创建脚本和使用鼠标快捷方式创建脚本文件，`touch`命令的使用将在第3章中进行介绍。

刚创建的文件的文件名是高亮状态，表示可以对文件按照需要进行重命名，将文件重命名后按下回车键，从而完成了脚本文件的创建。脚本文件名的命名方式要尽量能够描述脚本的作用，这样通过文件名就可以大体知道脚本的功能，从而提高Shell脚本的可读性。

在屏幕的空白区域右击鼠标，在弹出的快捷菜单中选择【创建文件】，这样就可以创建一个普通文件。文件创建后，可以通过双击鼠标左键打开文件，还可以使用右击鼠标在快捷菜单中选择【打开】菜单项打开文件。文件打开后如图2-8所示。

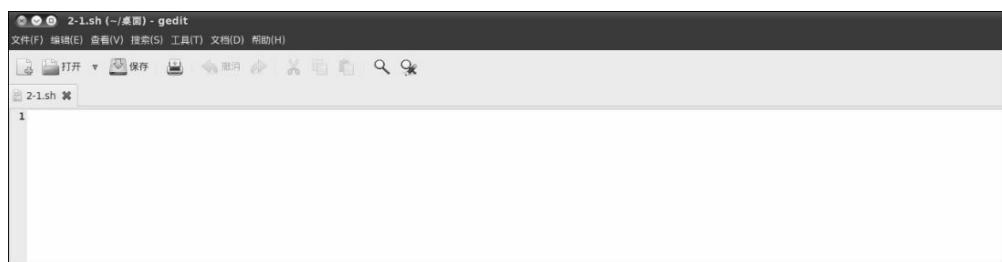


图2-8 刚打开的空白脚本文件示意图

如果系统中存在很多的文本编辑器，还可以选择默认的文本编辑器。选择默认的文本编辑器的方式是右击鼠标，在打开的快捷菜单中选择【属性】，在属性菜单中选择打开方式，然后选择合适的方式。笔者一般是选用gedit编辑器作为默认编辑器。选用方式如图2-9所示。



图2-9 设置默认属性

打开文件后，根据需要执行何种操作及实现何种功能来编写Shell脚本文件。编写脚本时要注意每一条语句在文件中要单独一行存在，而且在每一行的最后不需要添加任何符号，所以在一行中不要使用多个命令，防止Shell无法进行解析。如果想在一行中出现多个语句，那么就需要在语句之间使用分号进行分隔，防止Shell将多条语句看做一条语句，从而产生不必要的错误。

注意

在gedit文本编辑器中，Shell脚本中的不同内容会显示不同的颜色，可以通过颜色的不同来对脚本内容进行分辨。

在编写完脚本后，还需要赋予脚本可执行权限，否则脚本不能被Shell执行。在Linux系统中，任何需要执行的程序都要具有可执行权限。当脚本获得了可执行权限后，才能被Shell执行。

Shell脚本在执行之前，程序员不知道编写的脚本是否正确，只有在执行的过程中才能知道编写的Shell脚本是否正确。如果执行结果不正确，那么就需要根据执行结果进行适当的修改，直至达到最终的目标。

按照上面的流程创建示例脚本2-1. sh，其内容如下。

【示例脚本2-1. sh】

```
# 第一个Shell脚本，功能是输出字符串“Hello, Bash Shell”
#!/bin/bash
echo “Hello, Bash Shell”
echo
```

在上面的脚本中，第1句为注释部分，用来说明该脚本只是显示字符串“Hello, Bash Shell”；第2句用来表示使用/bin/bash来解释执行该脚本，即使用Bash解释执行脚本中的命令；第3句和第4句分别显示字符串“Hello, Bash Shell”和一个空行，从而在Shell命令行界面中显示字符串“Hello, Bash Shell”。

在首次使用gedit编辑器的时候，脚本文件中不会显示行号，但是编

辑器可以添加行号。gedit编辑器添加行号的方式是在gedit编辑器的【编辑】菜单中选择【首选项】，然后勾选【显示行号】选项，如图2-10所示。



图2-10 添加行号

单击【确定】按钮之后，就会在文本的左边显示行号。脚本编辑完成之后如图2-11所示。



图2-11 脚本编辑完成后的示意图

注意

在首选项中还可以设置使用4个空格代替TAB键，从而消除因TAB键表示的长度不同而造成脚本布局不美观的情况。

命令echo的作用是将命令后面的字符串输出到显示器上，一般用来显示提示性的信息或是命令或语句的执行结果等。除了能在显示器上显示结果以外，还可以在其他地方显示，这涉及Linux的重定向的内容，将在后面的章节中进行讲述。

在赋予脚本可执行权限后，运行脚本2-1. sh的结果如下：

```
ben@ben-laptop:~$ chmod u+x 2-1. sh
ben@ben-laptop:~$ ./2-1. sh
“Hello, Bash Shell”
```

```
ben@ben-laptop:~$
```

上面的脚本执行过程展示了脚本从创建到最终成功运行的整个过程，对于所有的新建脚本来说，都需要遵循上面的过程进行编写。而对于脚本中的一些特殊内容将在接下来的小节中进行详细介绍。

注意

关于echo命令的选项的使用方法，将在第4章中进行详细介绍。

2.3.2 Shell脚本中的格式介绍

在上面的示例脚本2-1. sh中出现了一些特殊的符号，这些符号在Shell脚本中被赋予了特殊的含义和作用。下面将对这些符号进行讲述。

1. 注释符号的使用

在【示例脚本2-1. sh】中的第一句为Shell脚本中的注释。注释的作用就是说明当前Shell脚本所实现的功能。注释行不会被Shell执行。使用注释对Shell脚本或其中的语句进行说明是非常必要的，否则在经过一段时间以后再看该脚本就会不清楚该脚本到底实现了什么功能。

2. 符号“#!”的使用

在每个Shell脚本的第一行几乎都使用符号“#!”开始，该符号并不是注释的符号，而是告知Shell终端执行当前Shell脚本需要使用哪种Shell，是Bash还是其他的Shell。如果没有这一行，一般的Shell会使用系统默认类型的Shell来执行该脚本。

注意

通过查看环境变量\$SHELL就可以获得系统的默认Shell类型。而对于大部分Linux系统来说，默认的Shell为Bash。

2.3.3 如何执行Shell程序

当Shell脚本编写完毕后就可以执行脚本，但是在执行脚本之前需要首先赋予这个脚本可执行权限，否则脚本不会执行。出现的错误如下所示：

```
ben@ben-laptop:~ $ ./2-1. sh  
-bash: ./2-1. sh: 权限不够  
ben@ben-laptop:~ $
```

出现这种错误的原因是此时的脚本文件还只是一个普通的文本文件，只是文件的内容是Shell脚本的内容，要想使脚本能够顺利执行，需要使用chmod命令赋予脚本可执行权限，如下所示：

```
ben@ben-laptop:~ $chmod u+x 2-1. sh
```

上面语句的作用是为脚本2-1. sh赋予可执行权限，但是只有用户本身具有可执行权限，其他的用户仍然不具有可执行权限，如果其他的用户也需要执行该脚本，那么就要根据实际需要赋予脚本文件不同的可执行权限。关于如何使用chmod命令赋予脚本权限，将在下一章中详细地介绍。

在赋予了脚本可执行权限后，就可以运行脚本，从而按照顺序执行脚本中的命令了。脚本运行的方式如下。

```
ben@ben-laptop:~ $ ./2-1. sh  
"Hello, Bash Shell"
```

ben@ben-laptop:~ \$方式使用的是“.”运算符来执行脚本，表示执行的脚本文件在当前的目录中，离开了当前的目录再使用“.”符号就不能执行脚本，如果强制执行，会提示找不到需要执行的脚本，如下所示：

```
ben@ben-laptop:~ $ ./2-1. sh  
bash: ./2-1. sh: 没有那个文件或目录  
ben@ben-laptop:~ $./2-1. sh  
"Hello, Bash Shell"  
  
ben@ben-laptop:~ $
```

在上面的执行结果中，第1次执行脚本时，在当前的目录中没有没有脚本文件2-1. sh，因此在运行时，bash会提示没有这个文件或目录。

而第2次运行时，是在2-1. sh文件所在的目录中运行的，因此能够正常运行。

第一种方式是使用系统默认的Shell执行脚本，除此之外，还可以使用直接指定特定的Shell来执行脚本。可以使用系统中存在的任意类型的Shell执行当前的脚本。但是也需要在脚本文件存在的路径下才可以执行，否则在执行脚本时也会提示找不到脚本文件。使用特定的Shell执行脚本如下：

```
ben@ben-laptop:~ $ sh test.sh  
"Hello, Bash Shell"  
  
ben@ben-laptop:~ $
```

在上面的实例中，第一句使用了sh来解释Shell脚本程序，在使用特定的Shell时，基本方式如下：

shell名称 脚本名称

除了使用sh来运行Shell脚本以外，还可以使用其他类型的Shell来运行脚本，如ksh、csh等。

注意

使用其他类型的Shell来执行脚本时，是按照当前使用的Shell类型来解释这些命令的，因此尽量使用通用的语法编写脚本，以防因为Shell类型不同而造成脚本不能正常执行。

如果想在任何的目录中都可以执行某个脚本，那么就需要使Shell能够找到这个脚本，这样就需要将脚本所在的路径放到环境变量中。关于如何更改环境变量，将在第4章中详细讲述。

2.4 小结

本章主要介绍了Shell脚本的基本入门知识，首先介绍了脚本语言的基本知识以及Shell终端的基本使用方式。然后对Shell命令的格式以及命令行进行了介绍。最后使用最简单的“Hello, Bash Shell”脚本介绍了Shell脚本的编写过程以及运行方式。

脚本语言在每一种操作系统中都存在，在Linux系统中，最常用的是Shell脚本。Shell脚本从根本上来说，就是将基本的Shell命令放到一块，使用各种控制结构将这些命令组合到一块，从而构成Shell脚本程序。当脚本执行时，就会按照Shell命令的先后顺序依次执行。

Shell命令需要在Shell终端中执行。在Shell终端打开后，在输入光标的前面存在一系列的字符，这些字符表示当前终端的一些信息，如当前使用的用户以及当前的路径，这些信息能够对脚本的正确运行起到辅助作用。

Shell脚本在编写时要按照一定的顺序进行，并且脚本的内容也需要按照特定的格式来进行编写。

第3章 Shell常用命令大演练

在编写Shell脚本时，Linux系统中的命令是脚本编写的基础，而且在Linux系统中常用的命令也是使用Linux系统的基础。因此熟悉常用的Shell命令既能编写出高质量的脚本程序，又能掌握Linux系统的使用。

本章的主要内容如下：

- 用户和用户组管理相关命令介绍
- 文件和目录操作相关命令介绍
- 系统管理相关命令介绍

3.1 Shell命令使用基础

Shell命令的使用在Linux系统基础之上，因此，在介绍Shell命令之前，首先介绍一些Linux系统的基础内容，如文件的基本知识等，这样可以更加深刻地了解Shell命令的功能及使用方式。

3.1.1 文件类型

在Linux系统中，所有的内容都被看做文件来处理，如在Windows系统中的文件夹在Linux系统中被称为目录文件，而常用的设备（如磁盘、串口等）也被当做文件来处理。这样可以简化系统对不同设备的处理，从而提高处理的速度。在Linux系统中，一般包括4类文件：普通文件、目录文件、设备文件和链接文件。

普通文件可以认为是文本文件，如Windows系统中的txt文件、word文件等，除此之外，在Linux系统中普通文件还包括二进制文件等。

目录文件就是Windows系统中的文件夹，目录文件不但包括本身的文件名以及其他属性，还包括存储在该目录文件中子文件的名称、大小等属性。因此，对目录文件的操作不会涉及文件的内容，而只是对文件名以及存储位置等进行各种操作。

设备文件用来表示Linux系统中的所有的硬件设备，这些硬件设备可以分为块设备和字符设备两种类型。块设备文件是指设备在读取时按照块为单位进行读取，如硬盘等。而字符设备则指代在读取时按照字符的顺序进行读取的设备，如键盘等。所有的设备文件都存储在/dev目录中，使用这种方式大大地方便了用户对硬件设备的直接操作。

链接文件类似于Windows系统中的快捷方式，但是链接文件比快捷方式功能更加强大，链接文件实现了对不同目录、不同系统中文件的直接访问，对于不同机器上的文件也可以使用链接文件进行直接访问。

链接文件按照其特性可以分为硬链接和软连接两种方式。硬链接引用的是源文件的物理索引，也可以称为节点。这就相当于为文件创建了一份备份文件，即使源文件被删除，通过硬链接文件也可以访问文件，直到所有的硬链接都被删除后，该文件才会被删除。而软连接引用的是文件的位置，类似于Windows系统中的快捷方式。当通过软链接访问文件时，实际上访问的还是源文件本身。当源文件被删除后，软连接文件随之失效，因此软链接又可称为符号链接。

注意

硬链接只能引用同一文件系统中的文件，而软链接不但可以引用不同文件系统中的文件，还可以引用远程机器上不同文件系统中的文件。

在Linux系统中使用一些特殊的符号表示不同的文件类型，这些符号以及对应的文件类型如表3-1所示。

表3-1 常用的文件类型及其符号表示

符号	文件类型	作用
-	普通文件	普通的文本文件，储存文本
d	目录文件	类似于文件夹，包含子目录文件
c	设备文件	对应Linux系统中的每一个硬件设备
l	链接文件	快捷方式文件

关于文件的详细信息，将在第6章中进行介绍。

3.1.2 绝对路径和相对路径

不管是对何种文件进行操作，首先要找到文件在系统中的存储位置，然后才能进行操作。因为对电脑来说，它不会自动去所有的目录文件中寻找需要操作的文件，而只会在当前的目录或指定的目录中进行寻

找。如果找不到需要操作的文件，就不会进行任何操作。

在Linux系统中一般采用绝对路径和相对路径两种方式表示文件的路径。绝对路径从根节点开始，直至找到本文件的路径。相对路径是以当前的路径为基准点，需要找到的文件相对于基准点的路径。关于绝对路径和相对路径的关系，可参见下面的示例。

```
/  
/home/ben  
/home/ben/test  
/home/ben/test/test.sh
```

对于上面列出的4个路径来说，第1个路径为Linux文件系统的根目录，第2个路径为ben用户的主目录，而第3个路径为目录文件test的路径，最后一个路径为文件test.sh的路径。上面的4个路径都是从根节点开始，因此这4个路径都是绝对路径。而通过第4个路径可以在系统任意的位置找到test.sh的位置。如果当前的目录为第2个目录，即在用户主目录中时，为了简单地表示test.sh的位置，可以使用相对路径表示。

使用相对路径表示文件位置时，一般是将当前的目录作为基准点，而需要确定位置的目录相对于基准点的位置就是该文件相对于当前目录的路径。为了能够找到test.sh，假设当前的路径为/home/ben，那么test.sh的相对位置就在test目录文件中，因此，其相对路径就为test/test.sh。而如果当前的目录为根目录时，test.sh的相对路径就变成了ben/test/test.sh。由此可见，相对路径不但和文件本身的位置相关，还与作为基准点的当前目录相关。

注意

在相对路径的前面不要添加反斜杠“/”，否则会被认为是Linux系统的根目录。

3.1.3 文件属性和文件权限

在Linux系统中，文件属性表示文件的一些基本的特性，如文件的节点、种类、权限模式、链接数量、所归属的用户和用户组、最近访问或修改的时间等内容，通过右击文件，在弹出的快捷菜单中选择属性，就可以显示文件的属性，也可以通过Linux命令ls获取，关于命令ls的使用将在下面的小节中详细介绍。

文件权限表示什么用户可以对文件进行何种操作。在Linux系统中，文件权限包括可读（使用字母r表示）、可写（使用字母w表示）、可执行（使用字母x表示）和没有任何权限（可以使用短线代替）共4种权限。用户所具有的对某个文件的权限可以是这3种权限的任意组合，可以只有一种权限，也可以具有3种权限，也可以没有任何权限。不管对文件进行何种操作，只有具有相应的权限才可以对文件进行相应的操作，没有权限的操作是不可以实现的。

对于普通文件来说，可读权限是具有此种权限的用户可以对文件进行读操作，可以读取文件中的内容。而可写操作是指用户可以对文件进行写操作，也就是可以向文件中添加某些内容，也可以删除文件中的某些内容。而对于可执行权限来说，比较直观的特点是双击可以运行。如

对于在第2章中出现的Shell脚本来说，既可以读取其中的内容，也可以对其进行修改，还可以运行该脚本，使其完成特定的功能。而对于目录文件来说，可读权限是指可以访问目录文件以及目录文件中的子文件，如查看文件的属性等，而可写权限则是表示可以在目录文件中创建新的文件，也可以删除其中的文件，而此时的可执行权限则表示允许进入目录文件的内部进行各种操作。

文件的权限除了使用特殊的字符表示之外，还可以使用数字来表示。在Linux系统中，一般将可读权限用数字4表示，可写权限用数字2表示，而可执行权限用数字1表示。如果不具有某项权限，那么该权限对应的数字就是0。文件所具有的权限是这3个数字的和，如果某个文件所具有的权限为可读可写，但是不能执行，那么该文件的权限可以记为“rw-”，还可以记为6。由此可见，使用数字表示文件的权限要比使用字符表示文件的权限方便得多，但是其可读性相对较差。因此，在实际使用时，可以参照实际的要求进行选择使用何种表示方式。

3.1.4 用户和用户组

用户和组的概念与学校里每一个学生和所在的班级之间的关系类似。在学校中，每一个学生都会有一个属于自己的小组，而每一个小组中也会包含很多的学生。在Linux系统中也包含许多的“学生”，这些“学生”就是可以自由使用Linux系统的用户，而许多用户可以组成一个用户组，这样在同一个用户组中的用户就享有操作Linux系统的同样的权限。

在每一个班级中都有一个花名册，用来标注该班级中的所有的学

生。而在Linux系统中，也包含这样类似的“花名册”。对于用户来说，这个“花名册”就是配置文件/etc/passwd和/etc/shadow，其中前者用来存储存在哪些用户，而后者用来存储登录系统时需要输入的密码。

1. /etc/passwd

/etc/passwd的部分内容如下：

```
ben@ben-laptop:~$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin) :/var/lib/g
```

```
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh  
libuuid:x:100:101::/var/lib/libuuid:/bin/sh  
syslog:x:101:103::/home/syslog:/bin/false  
messagebus:x:102:107::/var/run/dbus:/bin/false  
avahi-autoipd:x:103:110:Avahi autoip daemon,,,,:/var/lib/avahi-autoipd:/bin/false  
avahi:x:104:111:Avahi mDNS daemon,,,,:/var/run/avahi-daemon:/bin/false  
couchdb:x:105:113:CouchDB Administrator,,,,:/var/lib/couchdb:/bin/false  
speech-dispatcher:x:106:29:Speech Dispatcher,,,,:/var/run/speech-dispatcher:/bin/false  
usbmux:x:107:46:usbmux daemon,,,,:/home/usbmux:/bin/false  
haldaemon:x:108:114:Hardware abstraction layer,,,,:/var/run/haldaemon:/bin/false  
kernoops:x:109:65534:Kernel Oops Tracking Daemon,,,,:/:/bin/false  
pulse:x:110:115:PulseAudio daemon,,,,:/var/run/pulse:/bin/false  
rtkit:x:111:117:RealtimeKit,,,,:/proc:/bin/false  
saned:x:112:118::/home/saned:/bin/false  
hplip:x:113:7:HPLIP system user,,,,:/var/run/hplip:/bin/false  
gdm:x:114:120:Gnome Display Manager:/var/lib/gdm:/bin/false  
ben:x:1002:1002::/home/ben:/bin/sh
```

上面显示的是配置文件/etc/passwd中的内容，文件中的每一行代表一条用户信息。每一条信息由若干个字段组成，字段和字段之间使用冒号“：“隔开。其中第1个字段表示登录名，就是在登录时使用的用户名。第2个字段都是x，该字段用来表示该用户对应的密码，而密码存在于配置文件/etc/shadow中，因此在该文件中使用x进行表示，这个字符仅代表在该位置需要存储用户的密码，而并不指代其具体的密码内容。第3个字段表示该用户的用户标识，一般使用UID（User ID）表示，UID可以唯一地表示某个用户。第4个字段表示用户所在组的标识，一

般使用**GID**（Group ID）表示，**GID**可以唯一地表示某个用户所在的组。第5个字段用来表示用户名全称，这是可选的字段，可以根据实际的需要确定是否进行设置。如在gdm这个用户中，用户的全称是**Gnome Display Manager**，而ben这个用户没有设置全称，因此为空。第6个字段为用户的主目录所在位置如gdm用户的主目录为是/var/lib/gdm，而ben用户的主目录则是/home/ben。第7个字段表示用户所用**Shell**的类型，如果不需 要设定该字段，可以设置为/bin/false，而对于用户ben来说，使用的默认**Shell**为/bin/sh。

2. /etc/gshadow

对于用户组来说，这个“花名册”就是配置文件/etc/group和/etc/gshadow，前者表示存在哪些用户组以及用户组的基本信息，后者是前者的加密文件，用来保证其中的内容不会被任意更改。配置文件/etc/group的内容如下：

```
ben@ben-laptop:~$cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:ben
tty:x:5:
disk:x:6:
lp:x:7:
mail:x:8:
```

news:x:9:
uucp:x:10:
man:x:12:
proxy:x:13:
kmem:x:15:
dialout:x:20:ben
fax:x:21:
voice:x:22:
cdrom:x:24:ben
floppy:x:25:
tape:x:26:
sudo:x:27:
audio:x:29:pulse
dip:x:30:
www-data:x:33:
backup:x:34:
operator:x:37:
list:x:38:
irc:x:39:
src:x:40:
gnats:x:41:
shadow:x:42:
utmp:x:43:
video:x:44:
sasl:x:45:
plugdev:x:46:ben
staff:x:50:

```
games:x:60:  
users:x:100:  
nogroup:x:65534:  
libuuid:x:101:  
crontab:x:102:  
syslog:x:103:  
fuse:x:104:  
lpadmin:x:105:ben  
ssl-cert:x:106:  
messagebus:x:107:  
mlocate:x:108:  
ssh:x:109:  
avahi-autoipd:x:110:  
avahi:x:111:  
netdev:x:112:  
couchdb:x:113:  
haldaemon:x:114:  
pulse:x:115:  
pulse-access:x:116:  
rtkit:x:117:  
saned:x:118:  
admin:x:119:ben  
gdm:x:120:  
nopasswdlogin:x:121:  
ben:x:1000:  
sambashare:x:122:ben  
ben:x:1002:
```

```
testgroup:x:1003:
```

对于文件/etc/group来说，第一个字段就是用户名，而最后一个字段则是用户所属的组，同一个组中的用户具有相同的权限，而文件/etc/gshadow则表示一个密码文件，在用户登录时会访问这个文件的内容，如果转换后的密文和文件中记录的密文一样，那么用户才能登录。

3.1.5 特殊目录介绍

在Linux系统中，包含一些使用特殊符号表示的目录，这样可以非常方便地跳转到这些特殊的目录，这些目录包括根目录、用户主目录、当前目录和上一层目录。下面就分别介绍这4个目录及其表示方式。

根目录就是Linux系统的根目录，一般使用符号“/”表示，在根目录中包含Linux系统中的所有内容，类似于Windows系统中的“计算机”，可以在根目录中找到任何存在于系统中的文件。

用户主目录就是在目录“/home/”中和用户登录名一致的目录，这个目录是打开Shell终端后的默认目录，一般使用符号“～”表示。当前用户的主目录可以使用符号“～”表示，而如果表示其他用户的主目录，需要添加相应的用户名，这样就可以表示对应用户的主目录。如使用符号“～root”可以表示root用户的根目录。

注意

如果要访问其他用户主目录中的文件，必须具有root权限，否则不允许访问其他用户文件中的任何内容。

当前目录就是指当前操作所在的目录，一般使用符号“.”表示，当前目录可以通过命令pwd获取，而在使用时一般使用符号“.”表示，如在执行Shell脚本时的第一个字符就是表示需要执行的Shell脚本在当前的目录中。

上一层目录表示当前目录的上一层目录，一般使用符号“..”表示，如目录/home的上一层目录就是根目录，而目录/home/ben的上一层目录就是用户主目录/home。上层目录和当前目录一般不会显示，只有使用ls命令的-a选项时才会显示，如下面的语句所示：

```
ben@ben-laptop~/test /$ ls -a  
. .. test test1 test2
```

使用这些特殊符号表示特定的目录，可以减少书写时的字符数，并且使指令通俗易懂，提高可读性，如下列语句所示：

```
ben@ben-laptop~/test /$ls -a  
. .. test test1 test2  
ben@ben-laptop~/test /$ cd  
ben@ben-laptop :~/ $ cd ..  
ben@ben-laptop :/home$
```

注意

根目录没有上一层目录，只包含当前目录。

3.2 用户和用户组管理

Linux系统像一个精密的仪器在准确地运转。而这台机器的操作者就是Linux系统中的用户，多个用户还可以组成用户组。不同的用户具有对Linux系统不同的操作方式和操作权限，有的用户仅可以进行“读”操作，而有的用户却可以进行“读写”操作，这就需要熟练地对用户和用户组进行管理，从而使得Linux这台“机器”运转得更加顺畅。

3.2.1 用户管理常用命令

1. useradd/adduser命令

(1) 作用

useradd和adduser命令可以用于创建用户登录帐号或更新添加用户的信息。在Linux系统登录时，可以输入新添加的用户名或其他的已经存在的用户名，在输入了正确密码以后，才可以直接进入Linux系统。adduser命令和useradd命令的使用方式类似，因此仅使用useradd命令为例进行讲解。

(2) 格式

```
useradd [-c comment] [-d home_dir]
         [-e expire_date] [-f inactive_time]
         [-g initial_group] [-G group[,...]]
         [-m [-k skeleton_dir] | -M] [-s shell]
         [-u uid [ -o]] [-n] [-r] login

useradd -D [-g default_group] [-b default_home]
         [-f default_inactive] [-e default_expire_date]
         [-s default_shell]
```

(3) 使用说明

使用该命令时，需要在/home中创建名称为登录名的目录文件，作为新添加用户的主目录。可以使用-d选项指定一个目录为用户的主目录。当使用该用户登录后，用户的主目录就成为每一个打开的Shell终端的默认目录。使用useradd命令的常用选项如表3-2所示。

表3-2 useradd命令常用选项

选项	功能介绍
-name filename	搜索文件名中包含filename的文件
-uid n	搜索uid为n的文件
-size n	搜索大小为n的文件，n可以添加适当的单位
-amin n	搜索过去n分钟内访问过的文件
-atime n	搜索最近n天内未访问的文件

-user username

搜索用户名为username的文件

```
#示例脚本3-1. sh useradd/adduser命令的使用
ben@ben-laptop:~$ useradd ben
useradd: cannot lock /etc/passwd; try again later.
ben@ben-laptop:~$ sudo useradd ben
[sudo] password for ben:
ben@ben-laptop:~$
```

在添加用户时，只有root用户才存在添加用户的权限，其他用户不能添加新用户。使用户具有root权限的方式有两种，一种是使用命令sudo进行临时的权限提升，当命令执行完毕后，下一个命令仍然不具有root权限。另外一种是使用具有root权限的用户登录系统，这样所有的操作都具有root权限，而不需要在执行命令时添加sudo命令提升命令的执行权限。

注意

新添加的用户必须在设定密码后才能正常使用，否则不能正常使用。添加成功后，在配置文件/etc/passwd中就会出现相应的记录。

2. passwd命令

(1) 作用

设定或更改用户的密码。

(2) 格式

```
passwd [options] [LOGIN]
```

(3) 使用说明

使用passwd命令时可以使用选项，也可以不使用选项。如果不使用选项，则默认操作为更改用户的密码，如果不存在用户名，默认的操作为修改当前用户的密码。在更改密码时，需要输入两次密码，第一次和第二次的输入必须完全相同才能正确地进行用户密码设置。而在输入密码的过程中，不会像其他输入密码时会显示星号或其他字符，而是在终端看不到任何的输入信息。

注意

在输入密码时看不到输入的内容，使用read命令的一个特定选项就可以实现。关于read命令的使用将在下一章中进行讲述。

Passwd命令的常用选项如表3-3所示。

表3-3 passwd命令常用选项

选项	功能介绍
-d	删除密码

-s	列出密码相关的信息
-l	锁定用户的密码
-u	解锁被锁住的密码

```
#示例脚本3-2. sh passwd命令的使用
ben@ben-laptop:~$ passwd ben
passwd: 您不能查看或更改 ben 的密码信息。
ben@ben-laptop:~$ sudo passwd ben
输入新的 UNIX 密码:
重新输入新的 UNIX 密码:
passwd: 已成功更新密码
ben@ben-laptop:~$
```

通过上面的示例可以看出，在使用passwd进行密码的修改时，也需要root权限。否则不允许进行密码的修改。

注意

在每一个示例的最后都存在一行“ben@ben-laptop:~\$”，该行的作用是表明命令执行完毕，在后面的章节中也会使用类似的方式来进行说明。

3. userdel命令

(1) 作用

删除用户登录账号以及相关的信息。

(2) 格式

```
userdel [-r] login
```

(3) 使用说明

使用userdel命令可以实现用户的删除。在Linux系统登录时不能使用已删除的用户登录系统。在删除用户时，要保证删除的用户不是当前正在使用的用户。只有不被使用的用户才能删除成功。

使用userdel命令时一般只有-r选项，该选项的作用是将用户主目录中的内容一并删除，而在其他位置的相关内容也会被删除。

```
#示例脚本3-3. sh userdel命令的使用
ben@ben-laptop:~$ userdel ben
/usr/sbin/userdel: 只有 root 才能从系统中删除用户或组。
ben@ben-laptop:~$ sudo userdel ben
[sudo] password for ben:
正在删除用户 'ben' ...
警告: 组“ben”没有其他成员了。
完成。
ben@ben-laptop:~$
```

通过上面的示例可以看出，删除用户也需要root权限。当删除了用

户后，如果用户所在组没有其他的成员，系统将会进行提示，以防止误操作的发生。

注意

在执行所有的用户管理命令时都必须具有root权限，否则在命令执行时会提示操作被禁止。

3.2.2 用户组管理常用命令

在Linux系统中，用户一般是按照用户组的方式进行管理，而用户组的操作也包括用户组的创建、删除等基本的操作，下面将一一介绍用户组操作相关命令的使用。

1. groupadd命令

(1) 作用

添加用户组。

(2) 格式

```
groupadd [-g gid [-o]] [-r] [-f] groupname
```

(3) 使用说明

在使用groupadd命令时，可以不适用任何的选项，但是必须指定groupname。当命令执行完以后，groupname对应的用户组就存在于系统中，可以查看配置文件/etc/group是否存在和groupname相关的信息。

groupadd命令的常用选项如表3-4所示。

表3-4 groupadd命令常用选项

选项	功能介绍
-g	指定用户组标识gid
-o	重复使用用户组标识
-r	创建系统组
-f	强制创建已存在的组

表3-4的选项中，-g选项指定用户的组标识符（GID），也可以不使用该选项而由系统自动进行分配。

（4）使用实例

```
ben@ben-laptop:~$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:ben
tty:x:5:
```

```
disk:x:6:  
lp:x:7:  
mail:x:8:  
.....  
ben:x:1002:  
ben@ben-laptop:~$ sudo groupadd testgroup  
ben@ben-laptop:~$ cat /etc/group  
root:x:0:  
daemon:x:1:  
bin:x:2:  
sys:x:3:  
adm:x:4:ben  
tty:x:5:  
disk:x:6:  
lp:x:7:  
mail:x:8:  
.....  
ben:x:1002:  
testgroup:x:1003:
```

通过上面的示例可以看出，在创建新的用户组以前，配置文件/etc/group中包含当前系统中存在的用户组，而使用groupadd命令创建了新的用户组以后，配置文件/etc/group就会在文件的最后添加相应的信息。

2. **groupdel**命令

(1) 作用

删除特定的用户组。

(2) 格式

```
groupdel groupname
```

(3) 使用说明

使用groupdel命令删除指定的用户组时，不需要使用任何选项。而groupname既可以使用文字表示的用户组名，也可以使用用户组标识符gid。

(4) 使用实例

```
ben@ben-laptop:~$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:ben
tty:x:5:
disk:x:6:
lp:x:7:
mail:x:8:
.....
ben:x:1002:
```

```
ben@ben-laptop:~$ sudo groupdel testgroup
ben@ben-laptop:~$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:ben
tty:x:5:
disk:x:6:
lp:x:7:
mail:x:8:
.....
ben:x:1002:
```

3.2.3 其他常用命令

对于用户和用户组的相关命令，除了上面介绍的创建、修改、删除这些基本操作之外，还有很多常用的命令，在本小节中仅对其中使用比较频繁的命令进行介绍。至于其他命令的使用方法，读者可以通过阅读相关资料进行学习。

1. su命令

(1) 作用

更改登录用户。

(2) 格式

```
su [OPTION]... [-] [USER [ARG]...]
```

(3) 使用说明

使用su命令可以更改登录用户。如在使用普通用户登录后，对某些系统文件缺少操作权限，这时可以使用该命令更改登录用户为root，这样可以提高操作权限。

注意

更改用户时，同样也需要输入登录密码。

(4) 使用实例

```
ben@ben-laptop:~$ su root  
密码:  
>pwd  
/root  
>exit  
ben@ben-laptop:~$
```

2. sudo命令

(1) 作用

临时更改某条命令的执行权限，使其作为其他的用户执行当前的命令。但是除当前执行的命令以外，在执行其他的命令时，其权限以及用户不会发生改变。

(2) 格式

```
sudo -h | -K | -k | -L | -V

sudo -v [-AknS] [-a auth_type] [-p prompt]

sudo -l[1] [-AknS] [-a auth_type] [-g groupname|#gid] [-p pr
[-U username] [-u username|#uid] [command]

sudo [-AbEHnPS] [-a auth_type] [-C fd] [-c class|-]
[-g groupname|#gid] [-p prompt] [-r role] [-t type]
[-u username|#uid] [VAR=value] [-i | -s] [command]

sudo edit [-AnS] [-a auth_type] [-C fd] [-c class|-]
[-g groupname|#gid] [-p prompt] [-u username|#uid] file ...
```

(3) 使用说明

虽然sudo命令可以选择使用的选项非常多，而且也可以临时地作为其他用户进行各种操作，但是经常使用的是作为root用户执行命令，这样可以在执行命令时获得更高的权限。

关于sudo命令，在前面的示例中已经多次使用，因此不再列举示例。对sudo命令的使用不是很熟悉的读者可以参看前面的示例。

3. who命令

(1) 作用

显示登录用户的信息。

(2) 格式

```
who [OPTION]... [ FILE | ARG1 ARG2 ]
```

(3) 使用说明

使用who命令可以列举出当前是哪个用户登录使用系统，并且可以显示出用户的登录时间、运行级别等信息。

(4) 使用实例

```
ben@ben-laptop:~$ who
ben  tty7          2013-08-04 21:25  (:0)
ben  pts/0          2013-08-04 21:26  (:0.0)
ben@ben-laptop:~$ who -a
                  系统引导 2013-08-04 21:22
                  run-level 2  2013-08-04 21:22
LOGIN      tty4          2013-08-04 21:22      821  i
LOGIN      tty5          2013-08-04 21:22      827  i
```

LOGIN	tty2	2013-08-04 21:22	836 i
LOGIN	tty3	2013-08-04 21:22	837 i
LOGIN	tty6	2013-08-04 21:22	841 i
LOGIN	tty1	2013-08-04 21:22	1018 i
ben + tty7		2013-08-04 21:25 old	1712 (:0)
ben + pts/0		2013-08-04 21:26 .	1939 (:0.0)

4. which

(1) 作用

确定命令的路径。通俗的说法是查看命令的存储路径。

(2) 格式

```
which [-a] filename ...
```

(3) 使用说明

使用which命令时一般不需要使用任何选项，而只需要添加要查看的命令名即可准确定位。如果查找的命令不存在，那么会提示不存在该命令。

(4) 使用实例

```
ben@ben-laptop:~$ which which
/usr/bin/which
```

```
ben@ben-laptop:~$ which su  
/bin/su  
ben@ben-laptop:~$which 123  
ben@ben-laptop:~$
```

通过上面的执行结果可以看出，使用which命令能够发现执行的命令使用的到底是哪个命令，即对使用的命令进行定位，从而避免当存在多个相同的名称时，用户不知道执行的是哪一个命令。

注意

当命令不存在，或者是无法对需要定位的命令进行定位时，which命令不会显示任何的内容。

3.3 文件和目录操作

文件是Linux系统中的基本内容，在Linux系统中，所有的内容包括硬件设备都被认为是文件，因此，对于文件的操作就变得非常重要。

操作普通文件和目录文件时一般使用不同的命令，因此将其分开进行介绍。虽然在Linux系统中所有的内容都是文件，但是每一类文件具有不同的特点，因此不能采用同一种操作方式。

3.3.1 文件操作常用命令

在进行操作之前，必须要保证被操作的文件是存在的，不存在的文件是不允许被操作的。常用的文件操作包括创建文件、查看文件的内容、向文件中添加内容以及删除文件等，下面就一一介绍这些常用的命令。

注意

在本小节中所表述的文件为普通文件，即Windows中的文本文件，对于其他的文件操作将在后面的章节中进行介绍。

1. **touch**命令

(1) 作用

创建普通文本文档。

(2) 格式

```
touch [OPTION]... FILE...
```

(3) 使用说明

使用**touch**命令时，直接添加需要创建的文件的文件名即可实现文本文档的创建。当执行完该命令后，会在当前执行命令的目录文件中创建文件名为filename的文本文档。该命令还可以用来创建多个文件，在创建多个文件时，需要将filenamelist中的filename使用空格或逗号隔

开。

使用touch命令的常用选项如表3-5所示。

表3-5 touch命令常用选项

选项	功能介绍
-a	改变档案的读取时间记录
-m	改变档案的修改时间记录
-c	假如目的档案不存在，不会建立新的档案。与——no-create 的效果一样
-d	设定时间与日期，可以使用各种不同的格式

(4) 使用实例

```
ben@ben-laptop:~$ pwd  
/home/ben  
ben@ben-laptop:~$ touch test  
ben@ben-laptop:~$ ls  
test  
ben@ben-laptop:~$ touch test1 test2  
test test1 test2
```

文件当前的目录为/home/ben，当执行完创建文件test后，在目录中出现了文本文件test。当需要一次创建多个文件时，可以使用空格分开，用来区别不同的文件名。

2. cat命令

(1) 作用

查看文件中的内容，并将文件的内容在显示器上显示。

(2) 格式

cat [OPTION]... [FILE]...

(3) 使用说明

文件创建后，可以使用cat命令查看文件中的内容，而文件的内容会显示在当前的终端中。

注意

如果查看的文件不存在，那么在使用cat命令时会出现相应的提示。

(4) 使用实例

```
ben@ben-laptop:~$ cat test.sh
#第一个shell脚本
#!/bin/bash
echo "hello world"
echo
```

```
ben@ben-laptop:~$ cat 1  
cat: 1: 没有那个文件或目录
```

3. tail命令

(1) 作用

查看文件中最后的部分内容。

(2) 格式

```
tail [OPTION]... [FILE]...
```

(3) 使用说明

直接使用tail而不加选项可以直接查看文件倒数10行。如果需要显示特定的行数，要使用选项“+”或“-”表示显示的行数。“+num”一般表示显示从num行后的内容，而“-num”表示文件中的倒数num行。

(4) 使用实例

```
ben@ben-laptop:~$ tail /etc/passwd  
usbmux:x:107:46:usbmux daemon,,,:/home/usbmux:/bin/false  
haldaemon:x:108:114:Hardware abstraction layer,,,:/var/run/h  
kernoops:x:109:65534:Kernel Oops Tracking Daemon,,,:/bin/f  
pulse:x:110:115:PulseAudio daemon,,,:/var/run/pulse:/bin/fal  
rtkit:x:111:117:RealtimeKit,,,:/proc:/bin/false
```

```
saned:x:112:118::/home/saned:/bin/false
hplip:x:113:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gdm:x:114:120:Gnome Display Manager:/var/lib/gdm:/bin/false
ben:x:1000:1000:ben,,,:/home/ben:/bin/bash
ben:x:1002:1002::/home/ben:/bin/sh
ben@ben-laptop:~$ tail -5 /etc/passwd
saned:x:112:118::/home/saned:/bin/false
hplip:x:113:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gdm:x:114:120:Gnome Display Manager:/var/lib/gdm:/bin/false
ben:x:1000:1000:ben,,,:/home/ben:/bin/bash
ben:x:1002:1002::/home/ben:/bin/sh
```

4. more/ less命令

(1) 作用

more命令的作用是使文件分屏显示。less的作用与more命令相反，more命令的作用是最多显示多少行，而less命令的作用是至少显示多少行。两者在使用方式上是相同的，因此仅以more命令为例进行介绍。

(2) 格式

```
more [-dlfpcsu] [-num] [+pattern] [+linenum] [file ...]
```

(3) 使用说明

这两个命令一般用于当显示的内容超过整个屏幕，但是需要多屏才

能显示完全的情形。该命令一次只显示满屏幕的内容，如果需要显示的内容没有充满屏幕，那么就会被全部显示出来，而不会再分屏显示。当分屏显示时，在屏幕的下方显示“—more——”字样，用来提示后面还有内容。用户可以通过按键向上方向键、回车键查看后面的内容，而使用向下方向键查看前面的内容。如果想退出分屏显示，需要按下q键或ESC键。

more命令常用选项如表3-6所示。

表3-6 more命令常用选项

选项	功能介绍
-p	在显示下一屏之前清屏
-d	显示更加详细的提示性信息
-s	将连续的空白行作为一个空白行显示
-num	显示行号

(4) 使用实例

```
ben@ben-laptop:~$more -7 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
```

```
man:x:6:12:man:/var/cache/man:/bin/sh
```

5. wc命令

(1) 作用

wc命令是Linux系统中常用的文件操作命令，一般用来对文件中的字符进行计算。

(2) 格式

```
wc [选项] 待处理文件
```

(3) 使用说明

该命令根据使用选项的不同，来对待处理文件中的某项参数进行统计。如果不指定文件名，那么就从标准输入中获取待统计的信息。wc命令常用的选项如表3-7所示。

表3-7 wc命令的常用选项

选项	作用	备注
-c	显示字节数	同选项——bytes和-chars
-l	显示列数	同选项——lines
-w	显示字数	同选项——words
-L		无

	显示文件中最长行的长度	
-m	显示文件中的字符数	无

通过表3-7可以看出，wc命令使用不同的选项，就能获得文件中不同的信息。而在进行统计时，字是由空格字符区分开的最大字符串，每行结尾处的换行符也算一个字符，空格也算一个字符，而一个汉字被转换为3个字节。

(4) 使用实例

wc命令的使用方式如示例脚本3-1. sh所示。

【示例脚本3-1. sh】

```
#wc命令的使用方式
#!/bin/bash
echo 显示行数
wc -l 3-1.sh

echo 显示文件中最长行的长度
wc -L 3-1.sh

echo 显示Bytes数
wc -c 3-1.sh
```

对示例脚本3-1. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ # chmod u+x 3-1. sh
ben@ben-laptop: ~ # ./3-1. sh
显示行数
11 3-1. sh
显示文件中最长行的长度
27 3-1. sh
显示Bytes数
157 3-1. sh
```

通过脚本3-1. sh的运行结果可以看出，使用wc命令能够计算出文件中的行数、最长行的长度、字节数等各种基本信息。

注意

运行结果将根据使用的选项来进行不同的处理，在输出处理结果时，首先输出统计结果，然后显示操作的文件名。

3.3.2 目录操作常用命令

目录文件的操作也需要先创建，然后才能进行各种操作。常用的目录文件操作包括目录的创建、显示目录中的文件、目录的跳转、目录文件（包括普通文件）的复制/移动/重命名以及删除文件等。下面将逐个进行讲述。

1. **pwd**命令

(1) 作用

用于显示当前的操作所在的目录。

(2) 格式

```
pwd [OPTION]...
```

(3) 使用说明

该命令一般不需要使用任何参数，当执行该命令之后，会以绝对路径的方式显示当前操作所在的路径，该命令一般用来确定当前操作的位置。

(4) 使用实例

```
ben@ben-laptop:~$ pwd  
/home/ben
```

2. **mkdir**命令

(1) 作用

创建目录文件。

(2) 格式

```
mkdir [OPTION]... DIRECTORY...
```

(3) 使用说明

使用该命令可以创建一个或多个目录文件，当需要一次创建多个目录文件时，需要使用空格将不同的文件名分开。如果不指定目录文件的存储位置，那么新创建的目录文件的位置为当前目录，如果需要更改目录文件的存储位置，那么需要使用绝对路径或相对路径进行目录文件位置的重新确定。

(4) 使用实例

```
ben@ben-laptop:/home$ mkdir ben
ben@ben-laptop:~/test$ mkdir ben/test1 test2
ben@ben-laptop:~/test$ ls
ben  test2
ben@ben-laptop:~/test$ls ben
test1
```

注意

表示路径的字符串部分不允许出现空格，否则Shell会将其当做多个文件。

3. ls命令

(1) 作用

显示目录文件信息，包括文件的属性等。

(2) 格式

```
ls [OPTION]... [FILE]...
```

(3) 使用说明

ls命令是Linux系统中常用的命令，用于显示目录文件中的子文件列表，也可以递归列出子文件中的所有文件。常用的选项包括“-s”、“-a”、“-i”和“-l”，这4个选项分别用来显示目录文件中的全部文件（包括隐藏文件）、文件属性等详细信息。

(4) 使用实例

```
ben@ben-laptop:~/test$ ls -l
总计 8
drwxr-xr-x 2 ben ben 4096 2013-08-04 21:40 test1
drwxr-xr-x 2 ben ben 4096 2013-08-04 21:40 test2
ben@ben-laptop:~$ ls -sail /home
总用量 12
4194306 4 drwxr-xr-x 3 root root 4096 2013-08-19 22:09 .
2 4 drwxr-xr-x 26 root root 4096 2012-08-20 22:54 ..
4194307 4 drwxr-xr-x 39 ben ben 4096 2014-03-18 21:06 ben
```

在使用ls命令时，Shell终端会将不同类型的文件按照不同的颜色进

行显示，这样即使不使用-l选项，也可以看出在当前目录文件中的文件都属于什么类型，常用的文件类型和颜色的匹配关系如表3-8所示。

表3-8 文件类型和颜色的匹配关系

颜色种类	表示的文件类型
白色	普通文件
蓝色	目录文件
绿色	可执行文件
红色	压缩文件
浅蓝色	链接文件
黄色	设备文件
灰色	其他文件
红色闪烁	有问题的链接文件

通过表3-8可以看出，在Shell终端中使用了不同的颜色来表示不同的文件类型，而通过颜色就可以看出默认的文件类型。如图3-1所示（关于不同的颜色，请读者在终端上试一下，本图为黑白两色示意）。

```
ben@ben-laptop:~$ ls
3-1.sh      test      Yozo_Office  模板 文档 运行结果
3-2.sh      test.tar.gz 播放列表.pls  视频 下载 桌面
examples.desktop tmp      公共的    图片 音乐
ben@ben-laptop:~$ ls /dev
ati          mapper      rfkill    tty     tty38  tty51
audio        mcelog      root      tty0   tty39  tty52
block        mem         rtc       tty1   tty4   tty53
bsg          mixer      rtc0      tty10  tty40  urandom
bus          mixer1     scd0      tty11  tty41  usbmon0
cdrom        net         sda       tty12  tty42  usbmon1
cdrw        network_latency sda1      tty13  tty43  usbmon2
char        network_throughput sda2      tty14  tty44  v4l
```

图3-1 使用ls命令显示的文件颜色

通过终端操作可以看出，使用ls命令以后，不同的文件类型是由不同的颜色进行表示，因为没有改变默认的环境变量，因此在图3-1中显示的颜色和表3-8中的颜色与文件类型是相匹配的。

注意

使用ls命令显示的文件的颜色由环境变量LS_COLORS来控制，如果修改了这个变量，那么不同文件显示的颜色也会发生变化。

4. mv命令

(1) 作用

文件移动，还可以进行文件的重命名。既可以用于目录文件，也可以用于普通文件。

(2) 格式

```
mv [OPTION]... [-T] SOURCE DEST
```

```
mv [OPTION]... SOURCE... DIRECTORY  
mv [OPTION]... -t DIRECTORY SOURCE...
```

(3) 使用说明

使用mv命令进行普通文件移动时，不需要使用任何的选项。如果移动的是目录文件，那么需要添加-r选项，否则无法完成目录文件的移动。

(4) 使用实例

```
ben@ben-laptop:~/test$ mv test1 ben  
ben@ben-laptop:~/test$ ls  
test2 ben  
ben@ben-laptop:~/test$ mv ben/test1 test2ben  
ben@ben-laptop:~/test$ ls  
test2 test2ben ben  
ben@ben-laptop:~/test$ ls ben  
ben@ben-laptop:~/test$
```

5. cp命令

(1) 作用

文件复制命令，既可以用于目录文件，也可以用于普通文件。

(2) 格式

```
cp [OPTION]... [-T] SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
cp [OPTION]... -t DIRECTORY SOURCE...
```

(3) 使用说明

使用cp命令进行普通文件复制时，不需要使用任何的选项。如果复制的是目录文件，那么需要添加-r选项，否则无法完成目录文件的复制。

(4) 使用实例

```
ben@ben-laptop:~/test$ ls ben
ben@ben-laptop:~/test$ cp test* ben
cp: 略过目录 “test2”
cp: 略过目录 “test2ben”
ben@ben-laptop:~/test$ cp -rf test* ben
ben@ben-laptop:~/test$ ls
test2  test2ben  ben
ben@ben-laptop:~/test$ ls ben
test2  test2ben
ben@ben-laptop:~/test$
```

6. rm命令

(1) 作用

删除目录文件，既可以用于目录文件，也可以用于普通文件。

(2) 格式

```
rm [OPTION]... FILE...
```

(3) 使用说明

使用cp命令进行普通文件删除时，不需要使用任何的选项。如果删除的是目录文件，那么需要添加-r选项，这样可以递归地删除目录文件中的子文件，否则无法完成目录文件的复制。

(4) 使用实例

```
ben@ben-laptop:~ $ ls
test2  test2ben
ben@ben-laptop:~ $ rm *
rm: 无法删除 “test2”: 是一个目录
rm: 无法删除 “test2ben”: 是一个目录
ben@ben-laptop:~ $ rm -rf  test2
ben@ben-laptop:~ $ ls
test2ben
ben@ben-laptop:~ $
```

3.3.3 文件权限管理常用命令

1. chmod命令

(1) 作用

更改文件的访问权限，所适用的文件既包括普通文件，也包括目录文件。

(2) 格式

<code>chmod [role] [+ - =] [mode] filename</code>	符号模式
---	------

<code>chmod [mode] filename</code>	绝对模式
------------------------------------	------

(3) 使用说明

在使用chmod改变文件的访问权限时，可以使用符号模式，也可以使用绝对模式。使用符号模式就是使用符号表示文件权限，而绝对模式使用数字表示文件权限的每一个集合，一般来说，并且对于初学者来说，使用数字模式更加有效，而且系统也是使用这种方式表示文件权限的。

选项中的role由字母“u”（表示文件所有者）、“g”（表示和用户同用户组的其他用户）、“o”（表示其他用户）、“a”（表示所有用户）组合而成。操作符“+”、“-”、“=”则分别表示赋予、收回、赋予某个特定权限并且收回其他所有的权限，只能选择其中的一种。选项mode则表示文件具有的3种属性，即可读权限（r）、可写权限（w）和可执行权限（x）的任意组合。

使用chmod的绝对模式时需要将文件的属性使用八进制表示，并且

需要使用3个数字分别表示文件所有者、用户所有者同组的用户、其他用户相应的权限，最终的数字是3种权限的数字之和。

(4) 使用实例

```
ben@ben-laptop:~ $ touch test2
ben@ben-laptop:~ $ ls -l
总计 0
-rwxrwxrwx 1 ben ben 0 2013-08-04 21:45 test
-rwxrwxrwx 1 ben ben 0 2013-08-04 21:45 test1
-rw-r--r-- 1 ben ben 0 2013-08-04 21:46 test2
ben@ben-laptop:~ $ chmod g+w test2
ben@ben-laptop:~ $ ls -l
\总计 0
-rwxrwxrwx 1 ben ben 0 2013-08-04 21:45 test
-rwxrwxrwx 1 ben ben 0 2013-08-04 21:45 test1
-rw-rw-r-- 1 ben ben 0 2013-08-04 21:46 test2
ben@ben-laptop:~ $
```

2. chown命令

(1) 作用

将指定的文件所有者改变为指定的用户或组。

(2) 格式

```
chown [OPTION]... [OWNER][:[GROUP]] FILE...
chown [OPTION]... —reference=RFILE FILE...
```

(3) 使用说明

在使用chown命令时，选项-R表示递归地改变目录文件中所有的子目录以及子文件的所属用户或所属组。filename可以表示一个文件，也可以表示多个文件，当需要更改多个文件时，文件和文件之间使用空格隔开。

(4) 使用实例

```
ben@ben-laptop:~ $ chown root:root test2
chown: 正在更改 “test2”的所有者: 不允许的操作
ben@ben-laptop:~ $ sudo chown root:root test2
ben@ben-laptop:~ $ ls -l
总计 0
-rwxrwxrwx 1 ben ben 0 2013-08-04 21:45 test
-rwxrwxrwx 1 ben ben 0 2013-08-04 21:45 test1
-rw-rw-r— 1 root      root      0 2013-08-04 21:46 test2
ben@ben-laptop:~ $
```

3. file

(1) 作用

查看文件类型，为文件“验明正身”。

(2) 格式

```
file [-bchikLnNprsvz] [--mime-type] [--mime-encoding] [-f na  
[-F separator] [-m magicfiles] file  
file -C [-m magicfile]  
file [--help]
```

(3) 使用说明

file命令一般不需要使用参数，而直接添加需要确定类型的文件即可。当命令执行后会显示文件的基本特性，如属于什么类型的文件等。

(4) 使用实例

```
ben@ben-laptop:~ $ file test  
test: empty  
ben@ben-laptop:~ $ file test2  
test2: empty  
ben@ben-laptop:~ $ file /etc/passwd  
/etc/passwd: ASCII text  
ben@ben-laptop:~ $ sudo file /bin/cat  
/bin/cat: ELF 32-bit LSB executable, Intel 80386, version 1  
ben@ben-laptop:~ $
```

通过上面的执行结果可以看出，使用root权限和不使用root权限得到的文件信息是不同的，对于一般的情况来说，具有root权限的用户使用file命令得到的文件信息要比普通用户获取到的信息量要多。

3.3.4 查找文件常用命令

1. find命令

(1) 作用

在特定的目录中查找文件。

(2) 格式

```
find [-H] [-L] [-P] [-D debugopts] [-Olevel] [path...] [express]
```

(3) 使用说明

在使用find命令时，首先需要指定要搜寻文件的路径， keyword表示需要查找的关键词，常用的选项如表3-9所示。

表3-9 find命令常用选项

选项	功能介绍
-name filename	搜索文件名中包含filename的文件
-uid n	搜索uid为n的文件
-size n	搜索大小为n的文件，n可以添加适当的单位
-amin n	搜索过去n分钟内访问过的文件
-atime n	搜索最近n天未访问的文件

-user username

搜索用户名为username的文件

通过表3-9可以看出，find命令在使用选项之后，进行文件检索的功能将更加强大，可以直接按照文件的大小和用户编号进行检索，还可以按照用户名以及仅对某个时间段之内的文件进行检索。

(4) 使用实例

```
ben@ben-laptop:~ $ find /home test  
/home/ben  
test  
ben@ben-laptop:~ $
```

上面的示例表示如何在目录/home中查找文件名为test的文件，最终在目录中找到了文件test，并且表示出了是哪个目录中存在该文件。

2. grep命令

(1) 作用

查找符合特定表达式的字符或字符串。

(2) 格式

```
grep [OPTIONS] PATTERN [FILE...]  
grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]
```

(3) 使用说明

grep命令一般需要和正则表达式一起使用，正则表达式就是需要匹配的字符串，也就是目标字符串。

(4) 使用实例

```
ben@ben-laptop:~ $ ls -l | grep *1*
-rwxrwxrwx 1 ben ben 0 2013-08-04 21:45 test1
ben@ben-laptop:~ $
```

3.4 系统管理相关

对于Linux系统来说，系统管理是Linux系统操作中非常重要的部分。因为系统就像Linux系统的骨架，用来支撑基于系统的各种操作。编写良好的Shell脚本也离不开Linux系统管理的支持。

3.4.1 网络操作常用命令

1. ping命令

(1) 作用

向目标主机发送回应请求。

(2) 格式

```
ping [-LRUbdfnqrvVaAB] [-c count] [-i interval] [-l prelength]
      [pattern] [-s packetsize] [-t ttl] [-w deadline] [-F flowlabel]
      [interface] [-M hint] [-Q tos] [-S sndbuf] [-T timestamp option]
      [-t timeout] [hop ...] destination
```

(3) 使用说明

ping命令一般用来判定网络连接是否通畅。如果网络连接通畅，那么发送的数据就能到达目的主机，并且能够收到目的主机发送的应答消息。如果目的主机不存在或通信链路发生故障，那么就不会收到任何的数据。

(4) 使用实例

```
ben@ben-laptop:~ $ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56 (84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=4.48 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.178 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.082 ms
^C
--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.082/1.581/4.483/2.052 ms
ben@ben-laptop:~ $
```

使用ping命令一般不会自动退出，需要按下Ctrl+C键终止命令的运行，从而得出数据的传输信息。

2. ifconfig命令

(1) 作用

基本的网络配置命令。

(2) 格式

```
ifconfig [-v] [-a] [-s] [interface]
ifconfig [-v] interface [aftype] options | address ...
```

(3) 使用说明

ifconfig命令单独使用时可以用来显示当前的网络配置信息，而添加选项后可以对网络信息进行各种配置，如配置IP地址、默认网关等。

(4) 使用实例

```
ben@ben-laptop:~ $ ifconfig eth0 192.168.142.128
ben@ben-laptop:~ $ ifconfig
eth0      Link encap:以太网  硬件地址 00:0c:29:02:3e:a1
          inet 地址:192.168.142.128  广播:192.168.142.255  掩码:255.255.255.0
         inet6 地址: fe80::20c:29ff:fe02:3ea1/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST  MTU:1500  跳点数:1
             接收数据包:286 错误:0 丢弃:0 过载:0 帧数:0
             发送数据包:49 错误:0 丢弃:0 过载:0 载波:0
             碰撞:0  发送队列长度:1000
             接收字节:27411 (27.4 KB)    发送字节:6870 (6.8 KB)
```

```
中断:19 基本地址:0x2024
```

```
lo Link encap:本地环回  
    inet 地址:127.0.0.1 掩码:255.0.0.0  
        inet6 地址: ::1/128 Scope:Host  
            UP LOOPBACK RUNNING MTU:16436 跳点数:1  
            接收数据包:14 错误:0 丢弃:0 过载:0 帧数:0  
            发送数据包:14 错误:0 丢弃:0 过载:0 载波:0  
            碰撞:0 发送队列长度:0  
            接收字节:984 (984.0 B)    发送字节:984 (984.0 B)
```

3. route命令

(1) 作用

显示或配置路由表信息。

(2) 格式

```
route [-CFvnee]  
route [-v] [-A family] add [-net|-host] target [netmask Nm]  
[metric N] [mss M] [window W] [irtt I] [reject] [mod] [dyn]  
[reinstate] [[dev] If]  
route [-v] [-A family] del [-net|-host] target [gw Gw] [netmask  
Nm] [metric N] [[dev] If]  
route [-V] [--version] [-h] [--help]
```

(3) 使用说明

一般用来显示路由信息或进行路由表的配置。

(4) 使用实例

```
ben@ben-laptop:~ $ route
内核 IP 路由表

目标       网关   子网掩码   标志   跳点   引用   使用   接口
192.168.142.0   *   255.255.255.0   U        1        0        0    eth0
link-local      *   255.255.0.0   U        1000     0        0    eth0
default    192.168.142.2   0.0.0.0   UG       0        0        0    eth0
ben@ben-laptop:~ $
```

4. netstat命令

(1) 作用

用于显示各种网络相关信息，如网络连接、路由表、接口状态等。

(2) 格式

```
netstat [option]
```

(3) 使用说明

netstat命令可以单独使用，也可以使用特定的选项。当单独使用该命令时，其显示的内容可以分为两部分，一部分是Active Internet

connections，也可以称为有源TCP连接，另一部分是Active UNIX domain sockets，也可以称为有源Unix域套接口。对于这两部分内容的具体含义，读者可以不必关心。因为在使用netstat命令时，一般需要和特定的选项一起使用，从而显示特定的内容。

netstat命令的常用选项如表3-10所示。

表3-10 netstat命令常用选项

常用选项	功能介绍
-a (all)	显示所有选项， 默认不显示Listen相关
-c	每隔一个固定时间， 执行该netstat命令
-l	仅列出有在 Listen (监听) 的服务状态
-r	显示路由信息， 路由表
-s	按照协议进行统计
-t (tcp)	仅显示tcp相关的信息
-u (udp)	仅显示udp相关的信息

表3-10中列出了netstat命令常用的选项及其作用，可以根据自己的实际需要选择使用哪个选项。

(4) 使用实例

在Shell终端依次运行下列命令， netstat命令执行的结果如下所示。

```

ben@ben-laptop:~$ netstat -u
激活Internet连接 (w/o 服务器)

Proto Recv-Q Send-Q Local Address      Foreign Address     State
ben@ben-laptop:~$ netstat -r
内核 IP 路由表

Destination      Gateway   Genmask   Flags    MSS Window irtt Iface
192.168.0.0      *         255.255.255.0   U        0 0          0 eth1
link-local        *         255.255.0.0     U        0 0          0 eth1
default    vrouter  0.0.0.0   UG 0 0        0 eth1

ben@ben-laptop:~$
ben@ben-laptop:~$ netstat -t
激活Internet连接 (w/o 服务器)

Proto Recv-Q Send-Q Local Address      Foreign Address     State
tcp      0      344  ben-laptop:57018 37.61.54.158:www      ESTABLISHED
tcp      0      1    ben-laptop:38006 123.125.70.108:www      FIN_WAIT_1
tcp      0      0    ben-laptop:55366 219.239.88.110:www      ESTABLISHED
tcp      0      0    ben-laptop:55388 219.239.88.110:www      ESTABLISHED
tcp      0      0    ben-laptop:55376 219.239.88.110:www      ESTABLISHED
tcp      0      0    ben-laptop:55363 219.239.88.110:www      ESTABLISHED
tcp      0      0    ben-laptop:55386 219.239.88.110:www      ESTABLISHED
tcp      0      1    ben-laptop:53891 180.76.22.49:www      FIN_WAIT_1
tcp      0      0    ben-laptop:55374 219.239.88.110:www      ESTABLISHED
tcp      0      0    ben-laptop:55365 219.239.88.110:www      ESTABLISHED
tcp      0      0    ben-laptop:55385 219.239.88.110:www      ESTABLISHED
tcp      0      0    ben-laptop:55383 219.239.88.110:www      ESTABLISHED

^C

```

在使用netstat命令时，有时候会显示的内容很多，如使用-t选项时所示。此时可以与grep命令一起使用，从而只输出需要的部分，略过不需要的内容。

3.4.2 系统资源管理常用命令

1. ps命令

(1) 作用

静态显示当前的进程信息。

(2) 格式

```
ps [options]
```

(3) 使用说明

使用ps命令可以添加选项，也可以不添加选项，如果不使用选项，那么显示的结果为当前活动的进程，而添加选项后，会根据不同的选项而获取不同的结果。经常使用的选项组合为“a”、“u”、“x”，这3个选项分别显示系统中的全部进程、以用户为主进行显示、显示所有的程序，不区分终端。

(4) 使用实例

```
ben@ben-laptop:~ $ ps
```

```
PID TTY      TIME CMD
 1944 pts/0    00:00:01 bash
 2388 pts/0    00:00:00 ps
ben@ben-laptop:~ $ 
ben@ben-laptop:~ $ ps aux | more
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root  1  0.0  0.1    2800 1660 ? Ss   21:22  0:02 /sbin/init
root  2  0.0  0.0      0     0 ? S   21:22  0:00 [kthreadd]
root  3  0.0  0.0      0     0 ? S   21:22  0:00 [migration]
]
root  4  0.0  0.0      0     0 ? S   21:22  0:00 [ksftire]
]
root  5  0.0  0.0      0     0 ? S   21:22  0:00 [watchdog]
root  6  0.0  0.0      0     0 ? S   21:22  0:00 [migration]
]
root  7  0.0  0.0      0     0 ? S   21:22  0:00 [ksftire]
]
root  8  0.0  0.0      0     0 ? S   21:22  0:00 [watchdog]
root  9  0.0  0.0      0     0 ? S   21:22  0:00 [events/0]
root 10  0.0  0.0      0     0 ? S   21:22  0:00 [events/1]
root 11  0.0  0.0      0     0 ? S   21:22  0:00 [cpuset]
root 12  0.0  0.0      0     0 ? S   21:22  0:00 [khelper]
root 13  0.0  0.0      0     0 ? S   21:22  0:00 [netns]
root 14  0.0  0.0      0     0 ? S   21:22  0:00 [async/mem]
root 15  0.0  0.0      0     0 ? S   21:22  0:00 [pm]
```

2. top命令

(1) 作用

动态显示所有的进程。

(2) 格式

```
top -hv | -bcHisS -d delay -n iterations -p pid [, pid ...]
```

(3) 使用说明

top命令一般单独使用，命令执行后，屏幕每隔一定时间就会刷新一次，将不存在的进程信息去掉，并且添加新的进程信息。

(4) 使用实例

```
ben@ben-laptop:~ $ top
top - 12:38:33 up 50 days, 23:15, 7 users, load average: 60.00
Tasks: 984 total, 1 running, 983 sleeping, 0 stopped, 0
Cpu(s): 0.1%us, 0.2%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi
Mem: 98957812k total, 92327516k used, 6630296k free, 17270k
Swap: 33554424k total, 16k used, 33554408k free, 87178292k ca

      PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
 17960 app_usr 15 0 13428 1800 816 R 1.3 0.0 0:00.33
 18566 app_usr 15 0 46672 11m 7116 S 0.3 0.0 122:2.0
      1 root 15 0 10368 680 572 S 0.0 0.0 0:06.81
```

```

2  root     RT -5 0      0  0 S  0.0  0.0   0:02.34 m
3  root     34 19 0      0  0 S  0.0  0.0   0:00.25 ksoft

```

运行结果的最前面是统计信息区，这些信息依次是当前时间、当前系统运行的时间、当前登录用户数、系统负载（包括1分钟、5分钟、15分钟前CPU的负载的平均值）。

在统计区的下面是进程和CPU的信息，当存在多个CPU时，出现的内容可能会超过两行，第1行的内容包括进程总数、正在运行的进程数、睡眠的进程数、停止的进程数、僵尸进程数，第2行的内容包括CPU信息主要包括在过去1分钟、5分钟、15分钟的CPU占用率以及CPU空闲率等。第3行主要是内存的相关信息，主要涉及内存总量、使用量、空闲内存数、缓存数等信息，第4行表示交换分区的相关信息，包括交换分区总数、交换分的使用量、空闲交换分区数以及缓存数等。

在统计信息的下面是进程的相关信息，展现的内容如表3-11所示。

表3-11 top进程区字段含义

字段	含义	字段	含义
PID	进程号	USER	启动进程的用户
PR	优先级	NI	进程nice值
VIRT	进程使用的虚拟内存总量	RES	进程使用的、未被换出的物理内存大小
SHR	共享内存大小	S	进程状态

%MEM	内存占用率	%CPU	CPU占用率
TIME+	运行时间	COMMAND	运行的命令

进程的状态主要包含以下几种：

- D, 表示不可中断的睡眠状态。
- R, 表示运行状态。
- S, 表示睡眠状态。
- T, 表示跟踪/停止状态。
- Z, 表示僵尸进程状态。

3. **free**命令

(1) 作用

显示内存空间的使用情况。

(2) 格式

```
free [-b | -k | -m | -g] [-o] [-s delay] [-t] [-v]
```

(3) 使用说明

使用free可以显示系统的内存使用情况。

(4) 使用实例

```
ben@ben-laptop:~ $ free -m
```

```
        total  used   free   shared  buffers  cached
Mem:     1001    569    432      0      56    347
        -/+ buffers/cache: 165 836

Swap:    687    0 687

ben@ben-laptop:~ $ 
ben@ben-laptop:~ $ free -k
        total  use   dfree   shared  buffers  cached
Mem:1025992        583484  442508      0        58388    356148
        -/+ buffers/cache: 168948 857044

Swap:704504    0    704504

ben@ben-laptop:~ $ free
        total  used   free       shared  buffers  cached
Mem:1025992        583484  442508  0        58396    356144
        -/+ buffers/cache: 168944    857048

Swap:704504    0    704504

ben@ben-laptop:~ $
```

3.4.3 磁盘信息查看常用命令

1. df命令

(1) 作用

显示剩余磁盘空间。

(2) 格式

```
df [OPTION]... [FILE]...
```

(3) 使用说明

使用df命令可以查看所有磁盘（物理文件系统）的使用信息，这些信息包括已用空间、可用空间（即剩余空间）、挂载点等信息。df命令可以单独使用，也可以和选项一起使用。在单独使用时效果和选项-a类似，可以显示所有物理文件系统的使用信息。

(4) 使用实例

```
ben@ben-laptop:~ $ df -m
文件系统      1M-块 已用 可用  已用% 挂载点
/dev/sda1    14439   5268 8438   39%  /
none        497    1497     1%  /dev
none        501    1501     1%  /dev/shm
none        501    1501     1%  /var/run
none        501    0501     0%  /var/lock
none        501    0501     0%  /lib/init/rw
ben@ben-laptop:~ $
```

2. **fdisk**命令

(1) 作用

显示磁盘分区或磁盘信息命令。

(2) 格式

```
fdisk [-uc] [-b sectorsize] [-C cyls] [-H heads] [-S sects]
fdisk -l [-u] [device...]
fdisk -s partition...
fdisk -v
fdisk -h
```

(3) 使用说明

fdisk可以用于显示磁盘信息，如柱面信息、分区数等，还能对磁盘进行分区。在进行磁盘分区时，采用问答式界面，而不是图形化的处理，因此操作不是很方便，但是其功能和图形化的操作完全相同。

(4) 使用实例

```
ben@ben-laptop:~ $ fdisk -l
ben@ben-laptop:~ $ sudo fdisk -l
[sudo] password for ben:

Disk /dev/sda: 16.1 GB, 16106127360 bytes
255 heads, 63 sectors/track, 1958 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical) : 512 bytes / 512 bytes
I/O size (minimum/optimal) : 512 bytes / 512 bytes
Disk identifier: 0x00097ffe
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1	*	1	1871	15021056	83	Linux
/dev/sda2		1871	1958	704513	5	Extended
/dev/sda5		1871	1958	704512	82	Linux swap /

注意

使用fdisk命令时，一般使用-l选项来显示磁盘的详细信息，并且需要使用root用户进行操作，否则，显示的内容将为空。

3.5 小结

本章主要介绍了Shell命令以及在使用Shell命令时涉及的Linux系统的基础内容，Shell是Linux系统重要的组成部分，而Shell命令是Shell完成各种功能的基础，因此为了更好地进行Shell脚本的学习，必须熟练掌握Shell命令。

Shell命令的基础是Linux系统的基本知识，如Linux系统中的文件类型、文件属性和文件权限的含义、用户和用户组的概念以及根目录和用户主目录的关系。掌握了这些内容是学好Shell命令的基础。

编写Shell脚本可以近似地理解为将使用的Shell命令放到Shell脚本中执行，Shell命令是Shell脚本的基础。这些命令可以分为3类，即用户和用户组管理相关命令、文件和目录操作相关命令以及系统管理相关命令，在每一类中又包含许多命令。这些命令在使用时可以添加选项，也

可以单独使用。这就需要读者不断地使用这些命令，从而达到熟练使用的目的。

第4章 Shell脚本中的变量

在编写Shell脚本时，经常会遇到某些数值不确定的情形，如在功能不变的前提下，这次处理/home目录，而下次有可能就会处理其他的目录。这样，不可能为了简单的目录改变而重新编写脚本。此时就需要使用一个特殊的标记来表示这个目录，这个特殊标记就是Shell脚本中的变量。本章就重点介绍变量在Shell脚本中的使用方式。

本章的主要内容如下：

- 变量的简单使用，如何输入和输出变量。
- 特殊变量的使用。
- 环境变量的设定与使用。
- 特殊的变量数组和字符串的使用。

4.1 变量的简单使用

变量的使用是Shell脚本编程中重要的内容，可以根据情景的不同来确定变量的最终数值，而在编写脚本时，在需要实际值的地方使用变量名来代替，而在实际使用时，参与脚本执行的是变量的值，而不是变量名。本小节将重点介绍变量的简单实用以及变量的输入和输出。

4.1.1 变量的使用

在任何编程过程中，使用变量表示在编程过程中需要动态确定的

值。在Shell编程中，也可以使用变量表示一些在程序运行中可能会变化的量。一般的使用方式如下面的表达式所示：

```
变量名=变量值
```

在上面的表达式中，变量名用来表示在脚本中需要使用的变量，变量值表示变量具体的数值，在脚本中使用变量名的地方都将用变量值代替。等号（=）的两边不允许出现空格或其他的符号，否则会产生错误，如示例脚本4-1. sh所示。

```
#示例脚本4-1. sh 变量赋值的正确形式和错误形式

#!/bin/bash

echo 正确的变量赋值方式

num=10      #为变量num赋值为整数10
str="10"     #为变量num赋值为字符串"10"

echo 错误的变量赋值方式

error = "error"    #错误的赋值形式，等号两边出现空格
```

对示例脚本4-1. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ # chmod u+x4-1. sh
ben@ben-laptop: ~ # ./4-1. sh
./4-1. sh: line 7: error: 找不到命令
```

脚本4-1. sh中变量num、str分别被赋予了整型数值10和字符串“10”，而第3种方式则因为等号的两边出现了空格，所以会出现运行结果中的错误。因此在变量赋值时，等号的两边不能出现任何符号。

在Shell中也可以使用变量，与C/C++中的变量类似。变量的命名方式与C/C++中类似，组成变量名的字符只能是字符、数字和下划线，并且数组不能用于变量名的开头。但是不允许包括空字符或“:”、“#”、“=”字符，这些字符不能出现在变量名中。在Shell中，变量是大小写敏感的，如“abc”、“Abc”和“ABC”分别代表3个不同的变量名。

在Shell中使用的变量也有各自的作用域，在Shell编程中一般会出现3种类型的作用域，第1种是在Shell脚本中的变量，这类变量只能在Shell脚本中使用，在Shell脚本以外不能使用脚本中的变量；第2种是Shell终端中使用的局部变量，该类型的变量范围是从在变量声明之后到当前的终端关闭之前；最后一种是在Shell终端中使用的全局变量，这类变量在所有的终端和所有的脚本中都可以使用。

在Shell脚本中使用的变量和其他编程语言不相同，在其他的编程语言中，需要首先定义变量，只有定义后的变量才能使用。而对于Shell编程来说，使用变量之前不需要提前定义，直接使用即可。在Shell编程中，没有变量类型的定义，可以直接为变量赋予各种类型的数据，而且可以对同一个变量名赋予各种类型的变量值，如示例脚本4-2. sh所示。

```
#示例脚本4-2. sh 为同一个变量名赋予不同类型的数值
#!/bin/bash
```

```
echo为变量num赋值为整数值
```

```
num=10
```

```
echo $num
```

```
echo为变量num赋值为字符串
```

```
num ="10"
```

```
echo $num
```

```
echo为变量num赋值为浮点型数值1.2
```

```
num=1.2
```

```
echo $num
```

对示例脚本4-2. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:～ # chmod u+x4-2. sh
```

```
ben@ben-laptop:～ # ./4-2. sh
```

```
为变量num赋值为整数值
```

```
10
```

```
为变量num赋值为字符串
```

```
"10"
```

```
为变量num赋值为浮点型数值1.2
```

```
1.2
```

通过程序运行结果可以看出，在Shell脚本中使用变量，不需要向其他高级语言那样定义了变量以后才能够使用，而在Shell脚本中，可以直接使用变量，并且可以多次为变量赋值。而变量最终的数值是最后一次

赋值的结果。

4.1.2 变量的输入和输出

在Shell编程过程中，除了示例脚本4-1. sh和示例脚本4-2. sh所表示的直接赋值的情形以外，还可以采用间接赋值的方式为变量指定具体的数据。在间接赋值时，需要在脚本执行过程中为变量输入不同的数值。对变量赋值可以使用read命令。read命令的使用方式如下：

```
read [option] 变量名
```

可以单独使用read命令，也可以添加选项。关于一些常用选项的具体使用将在后面详细介绍。

变量在脚本的运行过程中，会产生各种中间变量，在脚本中可以使用echo命令输出这些中间变量的值，而需要在变量的前面添加符号“\$”，如示例脚本4-3. sh所示。

```
#示例脚本4-3. sh 变量的输入和输出
#!/bin/bash

echo -n 输入变量的值:
read num #为变量num赋值为整数10
echo 变量的值为$num
echo
```

```
echo -n 输入变量的值:  
read num #为变量num赋值为字符串"10"  
echo 变量的值为$num  
echo
```

```
echo -n 输入变量的值:  
read num #为变量num赋值浮点型数值1.2  
echo 变量的值为$num
```

对示例脚本4-3. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ # chmod u+x4-3.sh  
ben@ben-laptop: ~ # ./4-3.sh  
输入变量的值: 10  
10  
输入变量的值: "1 0"  
"10"  
输入变量的值: 1.2  
1.2
```

通过示例脚本4-3. sh的执行结果可以看出，在Shell编程中，变量不需要声明而可以直接使用。在变量的赋值过程中，脚本不关心变量值属于何种数据类型，而只是用变量值替换变量名，这样可以将变量的使用过程大大地简化。

read命令和echo命令都包含很多选项，这些选项可以使read命令和echo命令的用途更加广泛，下面将详细介绍read命令的常用选项。

read命令的一般使用方式是接收来自标准输入（即键盘）的输入，或其他文件描述符的输入（关于文件描述符的内容将在第6章中详细介绍）。在获取输入数据以后，将数据放入一个标准变量中，如示例脚本4-4. sh中read命令的使用。如果不指定变量名称，那么输入的数值将保存在环境变量\$REPLY中，如示例脚本4-4. sh所示。

```
#示例脚本4-4. sh 不指定变量名输入数据，使其保存在环境变量$REPLY中
#!/bin/bash

echo "指定变量num"
read num    #指定变量num接收数值
echo '$num='$num
echo "未指定变量"
read      #不指定变量名，使数据保存在环境变量$REPLY中
echo '$num='$num
echo '$REPLY'=$REPLY
```

对示例脚本4-4. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:～ # chmod u+x4-4. sh
ben@ben-laptop:～ # ./4-4. sh
“指定变量num”
10
$num=10

“未指定变量”
```

```
100  
$num=10  
$REPLY=100
```

除此之外，read命令还可以使用一些选项，以便在接收数据的过程中更加贴近于实际需要，常用的选项及其功能如表4-1所示。

表4-1 read命令常用选项以及功能

选项	功能描述
-p	允许在read命令行中直接指定一个提示，可以同时为多个变量赋值
-t	指定read命令等待输入的秒数。当计时满时，read命令返回一个非零状态，并且直接退出等待输入过程
-n	指定接受到的字符个数，当达到指定个数后就退出输入状态，不管有没有按下回车键
-s	使read命令中输入的数据不显示在监视器上

在默认状态下，read命令一次只能为一个变量赋值，而使用-p选项可以同时为多个变量赋值。在为多个变量赋值时，变量和变量之间需要使用逗号或空格隔开，从而区分多个变量。在赋值过程中，按照变量出现的先后顺序依次为每一个变量赋值，如示例脚本4-5. sh所示。

```
#示例脚本4-5. sh 使用-p选项为多个变量赋值  
#!/bin/bash
```

```
echo "使用-p选项为多个变量赋值"
read -p "输入3个数值: " num1 num2 num3 #为多个变量赋值
echo "输出第1个变量的值: "
echo "num1 = $num1
echo "输出第2个变量的值: "
echo "num2 = $num2
echo "输出第3个变量的值: "
echo "num3 = $num3
echo

echo "交换赋值顺序"
read -p "输入3个数值: " num3 num1 num2 #为多个变量赋值
echo "输出第1个变量的值: "
echo "num1 = $num1
echo "输出第2个变量的值: "
echo "num2 = $num2
echo "输出第3个变量的值: "
echo "num3 = $num3
```

对示例脚本4-5. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ # chmod u+x 4-5. sh
ben@ben-laptop:~ # ./4-5. sh
使用-p选项为多个变量赋值
输入3个数值: 10 20 30
输出第1个变量的值:
```

```
num1 = 10
输出第2个变量的值:
num2 = 20
输出第3个变量的值:
num3 = 30

交换赋值顺序
输入3个数值: 10 20 30
输出第1个变量的值:
num1 = 20
输出第2个变量的值:
num2 = 30
输出第3个变量的值:
num3 = 10
```

通过示例脚本4-5. sh的执行结果可知，read命令在同时为多个变量赋值时，按照变量出现的先后顺序对变量赋值，因此要特别注意变量出现的顺序。

在使用read命令为变量的赋值时，如果没有为变量赋值，那么将一直停留在等待数据输入的状态而不再执行其他的操作。如果用户一直没有输入数据，那么脚本将不再继续向下执行，这样显然是不符合实际情况的。在使用-t选项后就可以避免此类的问题。-t选项用来限定等待输入的时间，一般以秒为单位。当到达指定的时间后，read命令就直接退出输入状态，而继续执行后面的命令。如示例脚本4-6. sh所示。

```
#示例脚本4-6. sh 使用-t选项限定等待时间
```

```
#!/bin/bash

echo "设定等待时间为4秒"
read -t4 num1      #限定等待时间为4秒
echo "num1 = $num1
echo "不限定时间输入"
read num1      #不限定时间输入
echo "num1 = $num1
```

对示例脚本4-6. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ # chmod u+x 4-6. sh
ben@ben-laptop:~ # ./4-6. sh
设定等待时间为4秒
num1 =
不限定时间输入
10
num1 = 10
```

通过示例脚本4-6. sh的运行结果可知，使用了-t选项以后，在规定的时间内，不管是否输入数据，都会继续执行后面的命令，从而避免了无休止的等待。

当为变量赋予字符串类型的数值时，有时候会出现需要3个字符但是却输入了5个字符的情形。对于其他的编程语言来说，可以通过限定字符的个数来控制变量实际接收的字符个数。但是Shell脚本却不具备这个功能，不管输入多少个字符，变量都会直接接收。使用-n选项则可以

避免这种类型状况的发生。在该选项的后面添加一个整数，该整数用来限定变量实际接收的字符个数，如示例脚本4-7. sh所示。

```
#示例脚本4-7. sh 使用-n选项限定输入字符个数
#!/bin/bash

echo "使用-n选项限定输入字符个数"
read -n4 num1      #限定为4个字符
echo "num1 = $num1
echo "不限定字符个数，输入10个字符"
read num1          #不限定时间输入
echo "num1 = $num1"
```

对示例脚本4-7. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:～ # chmod u+x 4-7. sh
ben@ben-laptop:～# ./4-7. sh
使用-n选项限定输入字符个数
1234"num1 = "1234
不限定字符个数，输入10个字符
1234567890
num1 = 1234567890
```

通过脚本4-7. sh的运行结果可知，在不使用-n选项时，可以无限制地为变量输入数据，但是使用了-n选项后，则将变量实际接受的数据长度限定在一个范围之内，使得变量不会接受所有的输入内容。

在使用read命令输入数据时，输入的内容默认显示在终端上。然而在某些情况下，如在输入密码时，一般不需要显示输入的内容，此时可以使用-s选项，这样就可以使得输入的内容不会在终端上显示，如示例脚本4-8. sh所示。

```
#示例脚本4-8. sh 使用-s选项不显示输入的内容
#!/bin/bash

echo "使用-s选项不显示输入的内容"
read -s num1      #不显示输入的内容
echo "num1 = $num1
echo "正常显示输入内容"
read num1      #显示输入的内容
echo "num1 = $num1
```

对示例脚本4-8. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:～ # chmod u+x 4-8. sh
ben@ben-laptop:～ # ./4-8. sh
使用-s选项不显示输入的内容
num1 =12345
正常显示输入内容
12345
num1 =12345
```

通过脚本4-8. sh运行结果可知，使用了-s选项以后，输入的内容变

得不可见了，当不使用该选项时，仍然可以看到输入的内容。因此，在实际使用过程中，需要按照实际的需求确定是否使用-s选项。

注意

使用-s选项时，数据仍然显示，只是 read命令将文本颜色设置成与背景相同的颜色，从而使得用户无法看到输入的数据。

4.1.3 变量的输出命令echo

在Linux系统中一般使用echo命令来进行变量的输出操作。echo命令用于在标准输出上输出传递给echo命令的所有参数（即在echo命令后面的内容，选项参数除外），echo命令的使用方式如下：

```
echo [选项] 输出字符串
```

使用echo命令时，也可以使用多个选项，常用的选项如表4-2所示。

表4-2 echo命令常用选项

选项	作用	使用示例
-n	输出文字后不换行	echo -n "hello"
-e	输出某些特殊字符	echo -e

——help	显示帮助	echo ——help
——version	显示版本信息	echo ——version

echo命令默认在输出结束之后，会输出一个换行符，即回车键，而在某些时候却不需要这个换行符，可以使用-n选项取消换行符，从而使得下一个命令还能在当前行执行，如示例脚本4-9. sh所示。

```
#示例脚本4-9. sh 使用-n选项不输出换行符
#!/bin/bash

echo "不使用-n选项输出换行符"
echo "输入数值: "
read num1
echo "输入的数值为: "$num1

echo "使用-n选项不输出换行符"
echo -n "输入数值: "
read num1
echo "输入的数值为: "$num1
```

为脚本4-9. sh赋予可执行权限后执行脚本，执行结果如下：

```
ben@ben-laptop:～ # chmod u+x 4-9. sh
ben@ben-laptop:～ # ./4-9. sh
不使用-n选项输出换行符
输入数值:
```

10

输入的数值为: **10**

使用 -n 选项不输出换行符

输入数值: **10**

输入的数值为: **10**

通过示例脚本4-9. sh的执行结果可以看出，当使用了-n选项之后，输入的内容不再回显到屏幕上，而是和背景采用同样的颜色，从而使得输入的内容不可见。在Linux系统中，一般使用这种方式来作为密码的输入，从而避免密码外泄。

在使用echo命令的-e选项时，某些特殊字符不再显示字符本身，而是按照特殊的功能进行显示。这些特殊的字符以及作用如表4-3所示。

表4-3 特殊的字符及其作用

字符	作用
\a	发出警告声，一般是蜂鸣声
\b	删除前一个字符
\c	最后不加上换行符号
\f（或\v）	换行但光标仍旧停留在原来的位置
\n	换行且光标移至行首
\r	光标移至行首，但不换行
\t	插入tab

\\\	插入\字符
\nnn	插入nnn（八进制）所代表的ASCII字符

在使用-e选项时，如果字符串中出现了表4-3中的字符，那么就按照特殊功能进行显示，而不会再按照普通字符进行显示，如示例脚本4-10. sh所示。

```
#示例脚本4-10. sh 使用-e选项输出特殊字符
#!/bin/bash

echo -e warning:\a
echo -e \"this is shell script\"
echo "插入反斜杠"
echo -e this \\is
```

对示例脚本4-10. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:～ # chmod u+x 4-10. sh
ben@ben-laptop:～ # ./4-10. sh
warning:
"this is shell script"
this \\is
```

在前面的示例脚本中，输出的内容都是连续的字符，字符与字符之间没有间断，而在实际的使用过程中，一个字符串可能由多个子串组成，如字符串“hello World”就由子字符串“hello”、“World”以及中间的

空格组成，对于这种情况，需要使用引号将作为字符串的部分括起来，防止出现二义性，如示例脚本4-11. sh所示。

```
#示例脚本4-11. sh 使用-s选项不显示输入的内容
#!/bin/bash

echo hello world          #输出字符串
echo "hello world"
str= hello world
echo $str      #输出变量
str="hello world"
echo $str
```

对示例脚本4-11. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ # chmod u+x 4-11. sh
ben@ben-laptop:~ # ./4-11. sh
hello world
hello world
./4-11. sh: line 4: world: 找不到命令
hello world
```

通过脚本4-11. sh的执行结果可以看出，在直接使用echo命令进行输出时，有没有引号不受影响，但是输出变量的值时，如果不使用引号，那么echo命令会将空格后面的字符串作为一个单独的命令来进行处理，从而发生各种意想不到的错误。

注意

在Shell脚本中，引号包括单引号、双引号、倒引号3种形式，而用来说明字符串的引号一般为双引号。单引号将会有其他的用途。至于引号在Shell脚本中的使用方式，将在下一章中进行详细介绍。

在使用echo命令时，如果不添加任何的选项和输出内容，在单独执行一个echo命令以后，就会输出一个空行，即一个回车键，如示例脚本4-12. sh所示。

```
#示例脚本4-12. sh 使用-s选项不显示输入的内容
#!/bin/bash

echo hello world      #输出字符串
echo
echo "hello world"
echo
str= hello world
echo $str      #输出变量
echo
str="hello world"
echo $str
```

对示例脚本4-12. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:～ # chmod u+x 4-12. sh
```

```
ben@ben-laptop:~ #./4-12.sh
```

```
hello world
```

```
hello world
```

通过示例脚本4-12. sh的执行结果可以看出，使用echo命令可以在输出字符的最后输出一个换行符，而单独使用echo命令将会输出一个空行，该空行一般用来隔开上下两部分输出，从而提高输出内容的可读性。

注意

echo命令可以使用重定向的方式将输出内容显示到其他的文件中，而不显示到计算机的屏幕上。关于如何使用重定向，将在后面的章节中进行讲述。

4.2 Shell中特殊变量的使用

上一节中讲述的变量只能用于当前Shell或当前的脚本中，在此范围之外，变量不能被使用。而在Shell中，只使用这些局部变量显然是不符合实际需要的。在某些场合，如在记录位置参数以及其他环境变量时，这些变量在所有的脚本中都会用到，因此这些变量被称为全局变量。这些变量在Shell脚本中具有特殊的功能，因此，也可称这些变量是特殊变量。

注意

全局变量一般可以看做是环境变量，关于环境变量的内容将在下一小节中进行讲述。

4.2.1 位置参数介绍

在Shell编程中，每一个脚本也可以使用一些参数。在Shell编程中存在一类参数，这些参数按照出现的先后顺序不同而进行赋值，因此一般被称为位置参数，常用的位置参数如表4-4所示。

表4-4 Shell脚本中的位置参数

参数名称	功能介绍
\$0	脚本名称
\$1～\$9	脚本执行时输入的第1至第9个参数
\$#	输入的参数个数
\$?	脚本返回值
\$*	参数的具体内容

表4-4中列出了Shell中常用的位置参数，这些参数记录了在Shell脚本执行时输入参数的详细信息，其中\$#表示输入的参数的个数，\$0则表示脚本的名称，而\$1～\$9则表示在各个位置上出现的参数的具体内容。

如果在某个位置不存在参数，那么该符号表示的内容为空，如示例脚本 4-13. sh 所示。

```
#示例脚本4-13. sh 使用位置参数
#!/bin/bash

echo "执行该脚本共输入了 $# 个参数"
echo "脚本的名字为 $0"
echo "第一个参数为 $1"
echo "第1个参数为 ${1}"
echo "第2个参数为 ${2}"
echo "第3个参数为 ${3}"
echo "第4个参数为 ${4}"
echo "第5个参数为 ${5}"
echo "第6个参数为 ${6}"
echo "第7个参数为 ${7}"
echo "第8个参数为 ${8}"
echo "第9个参数为 ${9}"
echo 脚本中传入的参数是: ${*}
```

对示例脚本 4-13. sh 赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop: ~ #chmod u+x 4-13. sh
ben@ben-laptop: ~ #./4-13. sh hello world 1234 567
执行该脚本共输入了4个参数
脚本的名字为 ./4-13. sh
```

```
第一个参数为hello
第1个参数为hello
第2个参数为world
第3个参数为1234
第4个参数为567
第5个参数为
第6个参数为
第7个参数为
第8个参数为
第9个参数为
```

通过脚本4-13. sh运行结果可知，使用特定的符号可以记录在脚本运行时使用的参数，从而使得在脚本的运行过程中，使用这些参数来参与脚本的执行。

4.2.2 \$@和\$*的区别

变量\$@和\$*都能在一个变量中包含所有的命令行参数，然而这两个变量又存在着不同。变量\$*是将命令行中的所有参数都作为一个单词来处理，这个单词中包含命令、脚本在执行时的所有的参数。而变量\$@是将所有的输入参数看做多个对象，是所有参数的列表，而不是和\$*一样，将每一个参数作为一个对象来处理。在一般情况下，这两个变量是没有区别的，且在使用时都需要用双引号引起，然而在使用for结构时，这两个变量的区别一览无遗，如示例脚本4-14. sh所示。

```
#示例脚本4-14. sh $@和$*的区别

#!/bin/bash

echo 使用for结构处理变量$@

count=1

for tmpstr in "$@"
do
    echo 第$count个变量的值为: $tmpstr
    count=$((count + 1))
done

echo
```

echo使用for结构处理变量\$*

```
count=1

for tmpstr in "$*"
do
    echo 第$count个变量的值为: $tmpstr
    count=$((count + 1))
done
```

对示例脚本4-14. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ #chmod u+x 4-14. sh
ben@ben-laptop:~ #./4-14. sh hello world 1234 567
使用for结构处理变量hello world 1234 567
```

```
第1个变量的值为: hello
```

```
第2个变量的值为: world
```

```
第3个变量的值为: 1234
```

```
第4个变量的值为: 567
```

```
使用for结构处理变量hello world 1234 567
```

```
第1个变量的值为: hell
```

```
第2个变量的值为: wofld
```

```
第3个变量的值为: 1234
```

```
第4个变量的值为: 567
```

通过示例脚本4-14. sh的执行结果可以看出，变量\$@一般将传递的参数作为一个参数来处理；而变量\$*则类似于一个数据组合，将所有的参数都作为一个单独的变量来处理。

4.3 环境变量的使用

环境变量是每一种编程语言都会使用的变量，这些变量记录了程序在执行时的一些基本信息，如行号、运行时间、日期等，在Shell脚本的执行过程中，也存在一些用来存储有关Shell终端会话和工作环境的信息，这些信息被称为Shell的环境变量。环境变量可以用来识别用户账户、Shell基本特性以及其他需要存储的内容，本小节中将详细介绍关于环境变量的相关内容。

4.3.1 Shell中的环境变量

所有的环境变量都使用大写字母，这样便于和普通变量区分。在Shell终端输入env命令或使用printenv命令就可以显示当前有效的环境变量，如示例脚本4-15. sh所示。

```
#示例脚本4-15. sh  显示环境变量
#!/bin/bash

echo "使用命令env显示环境变量"
env
```

对示例脚本4-15. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ #chmod u+x 4-15. sh
ben@ben-laptop:~ #./4-15. sh
使用命令env显示环境变量
ORBIT_SOCKETDIR=/tmp/orbit-root
SSH_AGENT_PID=1954
SHELL=/bin/bash
TERM=xterm
XDG_SESSION_COOKIE=767d70e78e56d25cc8dd8e284f3cb5d0-13769197
WINDOWID=75497477
GNOME_KEYRING_CONTROL=/tmp/keyring-XfQjcA
GTK_MODULES=canberra-gtk-module
USER=root
LS_COLORS=rs=0:di=01;34:ln=01;36:hl=44;37:pi=40;33:so=01;35:0
LIBGL_DRIVERS_PATH=/usr/lib/fglrx/dri
```

```
SSH_AUTH_SOCK=/tmp/keyring-XfQjcA/ssh
SESSION_MANAGER=local/ben-laptop:@/tmp/.ICE-unix/1911,unix/ben-laptop[1911]
USERNAME=root
DEFAULTS_PATH=/usr/share/gconf/gnome.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-gnome:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
DESKTOP_SESSION=gnome
QT_IM_MODULE=xim
PWD=/root
XMODIFIERS=@im=ibus
GDM_KEYBOARD_LAYOUT=cn
LANG=zh_CN. UTF-8
GNOME_KEYRING_PID=1893
GDM_LANG=zh_CN. UTF-8
MANDATORY_PATH=/usr/share/gconf/gnome.mandatory.path
GDMSESSION=gnome
SPEECHD_PORT=6560
SHLVL=1
HOME=/root
LANGUAGE=zh_CN:zh
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=root
XDG_DATA_DIRS=/usr/share/gnome:/usr/local/share/:/usr/share/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-NxGyLJBynH,socket=/tmp/dbus-NxGyLJBynH
LESSOPEN=| /usr/bin/lesspipe %
DISPLAY=:1.0
GTK_IM_MODULE=ibus
```

```
LESSCLOSE=/usr/bin/lesspipe %s %s
XAUTHORITY=/var/run/gdm/auth-for-root-kx9ZfA/database
COLORTERM=gnome-terminal
_=~/usr/bin/env
```

脚本4-15. sh运行结果显示了所有的环境变量，这些环境变量在当前用户的任意地方都可以使用。在使用环境变量时，要注意变量名都是大写字符，并且在变量名前面要添加美元符号“\$”。这些环境变量的作用表4-5所示。

表4-5 常用环境变量及其作用

环境变量名称	环境变量作用
\$SHELL	系统默认Shell类型
\$TERM	终端类型
\$USER	用户名
\$LS_COLORS	颜色信息
\$LOGNAME	当前用户的登录名
\$LANGUAGE	当前终端使用的语言
\$HOME	当前用户的主目录
\$PWD	当前路径
\$PATH	环境变量路径

如果只需要显示某个环境变量，那么使用echo命令再加上变量名就可以显示该变量的值，如下所示。

```
ben@ben-laptop:~$ echo $HOME  
/home/ben  
ben@ben-laptop:~$ echo $SHELL  
/bin/bash  
ben@ben-laptop:~$ echo $PWD  
/home/ben
```

通过上面的操作可以看出，环境变量的使用类似于普通变量的使用，除了用于输出该变量的内容之外，还可以参与其他的操作，这些操作在后面的章节中会出现，请读者细心阅读。

在书写环境变量时，变量名一般都是用大写字符，从而与其他类型的变量进行区分。

4.3.2 环境变量的配置文件

在上面的示例中显示的环境变量都存储在配置文件profile中。环境变量主要存在于3个文件中，这3个文件分别是：

- /etc/profile
- \$HOME/.bash_profile
- \$HOME/.profile

其中/etc/profile文件是系统默认的Shell配置文件，系统中的每一个

用户登录时都需要启动该文件，另外两个文件是某个特定用户登录系统时的启动文件，因此可以使用这些配置文件对每一个用户进行定制的启动。这3个文件的启动顺序如图4-1所示。

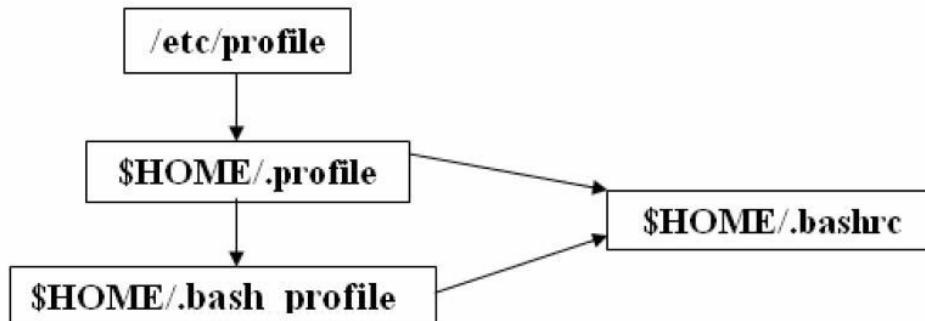


图4-1 环境变量相关配置文件执行顺序

文件/etc/profile是系统级文件，在用户登录时会自动执行该文件，该文件首先用来从/etc/profile.d目录的配置文件中搜集Shell的设置，可以设定Shell的命令提示符等，还可以设定掩码（umask）。该文件的内容如下：

```
ben@ben-laptop:~ #cat /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell
# and Bourne compatible shells (bash(1), ksh(1), ash(1))

if [ -d /etc/profile.d ]; then
    for i in /etc/profile.d/*.*sh; do
        if [ -r $i ]; then
            . $i
        fi
    done
```

```
unset i
fi

if [ "$PS1" ]; then
    if [ "$BASH" ]; then
        PS1='\u@\h:\w\$ '
        if [ -f /etc/bash.bashrc ]; then
            . /etc/bash.bashrc      fi
    else
        if [ "`id -u`" -eq 0 ]; then
            PS1='# '
        else
            PS1='$ '
        fi
    fi
fi

umask 022
```

当etc目录中的profile文件执行完以后，接下来要执行的文件就是主目录下的profile文件。profile文件是用户自己的配置文件，每一个用户需要配置的内容不同，profile文件的内容也就不同。而在该文件中，如果存在文件bashrc，那么就直接运行文件bashrc中的内容，来对用户进行设置。该文件的内容如下：

```
ben@ben-laptop:~ #cat $HOME/.profile
```

```
# ~/.profile: executed by Bourne-compatible login shells.

if [ "$BASH" ]; then
    if [ -f ~/.bashrc ]; then
        . ~/.bashrc
    fi
fi

mesg n
```

文件bash_profile也是每个用户自己“私有的”文件，这个文件用于配置用户自身使用的Shell信息。当用户登录时，该文件仅仅执行一次，默认情况下，该文件设置一些环境变量，执行用户的.bashrc文件。该文件的内容如下：

```
ben@ben-laptop:~$ cat .bashrc
# ~/.bashrc: executed by bash (1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the pac
# for examples

# If not running interactively, don't do anything
[ -z "$PS1" ] && return

# don't put duplicate lines in the history. See bash (1) for
# ... or force ignoredups and ignorespace
HISTCONTROL=ignoredups:ignorespace
```

```
# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in /etc/bashrc
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# make less more friendly for non-text input files, see lesspipe(1)
[ -x /usr/bin/lesspipe ] && eval "$(_SHELL=/bin/sh lesspipe)"

# set variable identifying the chroot you work in (used in the PWD)
if [ -z "$debian_chroot" ] && [ -r /etc/debian_chroot ]; then
    debian_chroot=$(cat /etc/debian_chroot)
fi

# set a fancy prompt (non-color, unless we know we "want" color)
case "$TERM" in
    xterm-color) color_prompt=yes;;
esac

# uncomment for a colored prompt, if the terminal has the capability
# off by default to not distract the user: the focus in a terminal
# should be on the output of commands, not on the prompt
```

```
#force_color_prompt=yes

if [ -n "$force_color_prompt" ]; then
    if [ -x /usr/bin/tput ] && tput setaf 1 >&/dev/null; then
        # We have color support; assume it's compliant with Ecma-48
        # (ISO/IEC-6429). (Lack of such support is extremely rare)
        # a case would tend to support setf rather than setaf.)
        color_prompt=yes
    else
        color_prompt=
    fi
fi

if [ "$color_prompt" = yes ]; then
    PS1='${debian_chroot:+($debian_chroot)}[\e[01;32m]\u@\h'
else
    PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
fi
unset color_prompt force_color_prompt

# If this is an xterm set the title to user@host:dir
case "$TERM" in
xterm*|rxvt*)
    PS1="\[\e[0;${debian_chroot:+($debian_chroot)}\u@\h: \w"
    ;;
*)
    PS1="\[\e[0;${debian_chroot:+($debian_chroot)}\u@\h: \w"
    ;;
esac
```

```
;;
esac

# enable color support of ls and also add handy aliases
if [ -x /usr/bin/dircolors ]; then
    test -r ~/.dircolors && eval "$(dircolors -b ~/.dircolors)$(alias ls='ls --color=auto'
#alias dir='dir --color=auto'
#alias vdir='vdir --color=auto'

alias grep='grep --color=auto'
alias fgrep='fgrep --color=auto'
alias egrep='egrep --color=auto'

fi

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Alias definitions.
# You may want to put all your additions into a separate file
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package

if [ -f ~/.bash_aliases ]; then
. ~/.bash_aliases
```

```
fi

# enable programmable completion features (you don't need to
# this, if it's already enabled in /etc/bash.bashrc and /etc/
# sources /etc/bash.bashrc) .

if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
    . /etc/bash_completion

fi
```

在该文件中设定了大部分常用的环境变量并且为这些变量赋值，如设定了记录的命令个数为1000个，这样就可以使用方向键来查看使用的命令历史。还对几个命令进行了重命名。当该文件被执行完以后，系统的控制权就交给其他的操作，从而执行后续的操作。

这3个脚本都是与环境变量相关的文件，并且都是在开机就会伴随着操作系统的运行而自动运行的，当脚本执行完毕之后，系统也随之启动成功。用户就可以在需要使用的地方使用环境变量了。

4.3.3 全局环境变量和本地环境变量

在Linux系统中，比较重要的变量是环境变量。环境变量定义了一些关于Linux操作系统以及当前使用的Shell终端的基本信息，显示全局环境变量一般使用env命令，而显示本地环境变量则使用set命令。使用这两个命令时，一般不需要使用其他的选项，如下面的代码所示。

```
ben@ben-laptop:~$ env
ORBIT_SOCKETDIR=/tmp/orbit-ben
SSH_AGENT_PID=1572
SHELL=/bin/bash
TERM=xterm
XDG_SESSION_COOKIE=767d70e78e56d25cc8dd8e284f3cb5d0-13938610
WINDOWID=77598880
GNOME_KEYRING_CONTROL=/tmp/keyring-etY5wo
GTK_MODULES=canberra-gtk-module
USER=ben
LS_COLORS=rs=0:di=01;34:ln=01;36:hl=44;37:pi=40;33:so=01;35:0
ng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m
LIBGL_DRIVERS_PATH=/usr/lib/fglrx/dri
SSH_AUTH_SOCK=/tmp/keyring-etY5wo/ssh
SESSION_MANAGER=local/ben-laptop:@/tmp/.ICE-unix/1535,unix/b
USERNAME=ben
DEFAULTS_PATH=/usr/share/gconf/gnome.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-gnome:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
DESKTOP_SESSION=gnome
QT_IM_MODULE=xim
PWD=/home/ben
XMODIFIERS=@im=ibus
GDM_KEYBOARD_LAYOUT=cn
LANG=zh_CN. utf8
GNOME_KEYRING_PID=1517
```

```
GDM_LANG=zh_CN. utf8
MANDATORY_PATH=/usr/share/gconf/gnome.mandatory.path
GDMSESSION=gnome
SPEECHD_PORT=7560
SHLVL=1
HOME=/home/ben
LANGUAGE=zh_CN:zh
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=ben
XDG_DATA_DIRS=/usr/share/gnome:/usr/local/share/:/usr/share/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-BiwpWHmQqb,
LESSOPEN=| /usr/bin/lesspipe %s
DISPLAY=:0.0
GTK_IM_MODULE=ibus
LESSCLOSE=/usr/bin/lesspipe %s %
XAUTHORITY=/var/run/gdm/auth-for-ben-zNuF6P/database
COLORTERM=gnome-terminal
_==/usr/bin/env OLDPWD=/

```

env命令显示了该系统中所有的环境变量，在使用时可以直接使用变量名来标识相应的环境变量。

注意

除了使用env命令之外，还可以使用printenv命令输出环境变量。

4.3.4 环境变量的设定

除了系统设定的环境变量以外，为了便于用户的使用，还可以由用户自己设定属于自己的环境变量。而在Bash Shell中，环境变量可以分为全局环境变量和局部环境变量，全局环境变量不仅在当前的Shell中有效，在所有Shell创建的子进程以及子Shell中都是可以被直接使用的。局部环境变量只能用于当前的Shell中，在其他的Shell中无法使用。

对于本地环境变量来说，可以使用set命令来显示所有的本地环境变量，全局环境变量一般使用export命令对变量进行设定，而使用env命令来显示环境变量的值。关于环境变量的设定如下所示。

```
ben@ben-laptop:~$ export MYHOME=/home/test  
ben@ben-laptop:~$ echo $MYHOME  
/home/test
```

如果想要使设定的变量在所有的Shell脚本中都可以使用，并且在开机之后就能使用，那么需要将变量放到开机运行的任意配置文件/etc/profile、\$HOME/.bash_profile和\$HOME/.profile中，在重启系统之后，就可以使用这个变量了。如果想在所有的用户中都可以使用设定的变量，那么就需要放入/etc/profile中，而其他的两个文件中的变量仅对于当前的用户起作用。如果想要该变量立即生效，那么需要使用source命令，使得配置文件立即生效，如下面的操作所示：

```
ben@ben-laptop:tail -f $HOME/.profile  
# ~/.profile: executed by Bourne-compatible login shells.
```

```
if [ "$BASH" ]; then
    if [ -f ~/.bashrc ]; then
        . ~/.bashrc
    fi
fi

mesg n

export MYHOME=/home/test
ben@ben-laptop:echo $MYHOME
ben@ben-laptop:
ben@ben-laptop: source $HOME/.profile
ben@ben-laptop:echo $MYHOME
/home/test
```

系统重启后再次使用变量MYHOME，输出内容如下：

```
ben@ben-laptop:echo $MYHOME
/home/test
```

通过上面的运行内容可以看出，在配置文件中添加了相应的变量以后，变量生效后就可以使用，而变量生效的方式有以下两种：

- 使用**source**命令立即生效
- 重启后生效

注意

设定的新的环境变量可以放在任何一个和环境变量相关的配置文件中，但是要注意它们的启动顺序，防止出现使用未定义的环境变量。

4.3.5 环境变量的取消

如果不再使用某个环境变量，可以使用unset命令取消该环境变量，使用unset命令时，只需要添加要取消的变量名即可，如下所示。

```
ben@ben-laptop:~$ export MYHOME=/home/test
ben@ben-laptop:~$ echo $MYHOME
/home/test
ben@ben-laptop:~$ unset MYHOME
ben@ben-laptop:~$ echo $MYHOME

ben@ben-laptop:~$
```

通过上面的操作可以看出，在进行了环境变量的取消操作之后，再次使用该变量时，该变量的值变成了空值，没有任何的意义。

4.4 小结

本章主要介绍了Shell脚本中变量的使用。变量用来表示一些不确定的内容，而按照变量的作用范围还可以将其分为局部变量和全局变量，全局变量也可以分为普通的全局变量和环境变量。

在Linux系统中，输入变量可以使用read命令，而变量值的输出一般使用echo命令。这两个命令一般都会按照实际的需要来实现某些特殊的功能，如输出一些特殊功能的字符、输入文本和背景颜色相同等。

在Shell脚本中，比较重要的是特殊变量和环境变量。特殊变量能够记录脚本运行时的一些属性信息，包括传递的参数个数、具体的参数信息等。而环境变量则可以在任何Shell脚本中使用。环境变量用来记录Shell脚本程序以及Shell终端在执行时的一些基本信息，如行号、运行时间、日期等基本信息。可以使用export命令和set命令来设定环境变量，使用unset命令来取消环境变量。

第5章 Shell脚本中的特殊符号

在上一章中讲述了Shell脚本中变量的使用方式，使用变量可以非常方便地表示那些在脚本执行过程中不断变化的量。本章将介绍Shell脚本中出现的特殊符号以及这些符号在Shell脚本中的作用。

本章的主要知识如下：

- 引号在Shell脚本中的应用，包括单引号、双引号和倒引号
- 通配符和元字符的使用
- 管道的使用
- 其他特殊字符的使用，如后台运行符、括号、分号等

注意

本章中介绍的特殊符号只是一些在Shell脚本中常用的特殊符号，除此之外，还会有其他的特殊符号，至于其他的符号本章中将不再讲述，如果后续章节中出现了某个特殊字符，将在出现的地方进行介绍。

5.1 引号的使用

在使用echo命令输出一个字符串时，需要使用引号将字符串括起来，从而使引号中间的部分都是字符串的内容，以避免歧义的产生。在Shell脚本中，引号一般包含以下3类。

- 单引号
- 双引号
- 倒引号

这3种引号除了可连接字符串以外（倒引号没有这种作用），还有其他的特殊用途，下面将分别进行讲述。

注意

所有的引号都需要两个相同的引号配套使用，单个引号将被视作单个字符处理，而不是引号。

5.1.1 单引号的使用

在编写Shell脚本时，Shell将美元符号“\$”、反斜杠“\”等符号作为特殊符号处理，然而在有些时候需要输出这些特殊符号，此时，需要使用单引号将这些特殊字符作为普通字符处理。而单引号的主要作用就是将单引号中的所有内容都作为普通的字符处理，即使里面的符号是特殊符号，如示例脚本5-1. sh所示。

```
#示例脚本5-1. sh 使用单引号输出特殊字符
#!/bin/bash

echo '不使用单引号'
echo '输出环境变量$HOME'
```

```
echo $HOME          #输出环境变量$HOME  
echo '使用反斜线控制符'  
echo a\tb\a\tc      #用反斜线控制符  
echo "输入一个反斜线"    #输入一个反斜线  
echo \\  
  
echo '使用单引号输出相应的符号'  
echo '$HOME'  
echo 'a\tb\a\tc'  
echo '\\'
```

对示例脚本5-1. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-1. sh  
ben@ben-laptop:~ $ ./ 5-1. sh  
'不使用单引号'  
'输出环境变量/home/ben  
/home/ben  
'使用反斜线控制符'  
atbatc  
"输入一个反斜线"  
\  
'使用单引号输出相应的符号'  
'/home/ben'  
'atbatc'  
'\\'
```

通过脚本5-1. sh的执行结果可以看出，在添加单引号以后，所有的字符都变成了普通的字符，原来被赋予的特殊含义已经不再存在。

注意

除了使用单引号之外，还可以使用转义字符将特殊字符转义为普通的字符。关于转义字符的使用方法将在后面的章节中进行讲解。

5.1.2 双引号的使用

双引号的作用和单引号的作用类似，都用于字符串的输出。但是符号\$、倒引号（`）和反斜线（\）仍作为特殊符号对待，而其余字符则被作为普通字符对待。使用双引号可以将某些Shell命令的输出内容作为字符串的一部分输出，这样就使得Shell脚本的输出内容更加丰富，如示例脚本5-2. sh所示。

```
#示例脚本5-2. sh 使用双引号输出特殊字符
#!/bin/bash

echo '不使用双引号'
echo '输出环境变量$HOME'
echo $HOME          #输出环境变量$HOME
echo '使用反斜线控制符'
echo a\tb\ta\tc      #用反斜线控制符
echo "输入一个反斜线"    #输入一个反斜线
```

```
echo \\

echo '使用双引号输出相应的符号'
echo "$HOME"
echo "a\`b\`a\`c"
echo "\\\"
```

对示例脚本5-2. sh执行权限后，执行结果如下：

```
ben@ben-laptop:～ $chmod u+x 5-2. sh
ben@ben-laptop:～ $ ./ 5-2. sh
'不使用双引号'
'输出环境变量/home/ben
/ben
'使用反斜线控制符'
atbatc
"输入一个反斜线"
\
'使用双引号输出相应的符号'
/ben
a\`b\`a\`c
\
```

通过脚本5-2. sh的执行结果可以看出，符号\$、反斜线(\)等特殊符号仍然被作为特殊符号处理，而除此之外的符号都被视作普通的字符。

5.1.3 倒引号的使用

倒引号就是键盘上数字键1左边的按键，直接按下该键就可以输出一个倒引号，而倒引号和其他的引号一样，需要两个引号配和使用。

倒引号的作用是以引号中的命令执行结果代替命令本身，这样在Shell脚本中就可以直接对命令的执行结果进行操作，如示例脚本5-3.sh所示。

```
#示例脚本5-3. sh 使用倒引号
#!/bin/bash

echo "使用倒引号封装命令"
echo """登录系统的用户为: "`who`"
echo "当前的时间为: " `date`"
echo "当前的文件的绝对路径为: " `pwd`"
```

对示例脚本5-3. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-3. sh
ben@ben-laptop:~ $ ./ 5-3. sh
使用倒引号封装命令
登录系统的用户为:
ben      tty7          2014-02-28 22:42  (:0)
ben      pts/0          2014-02-28 22:49  (:0.0)
ben      pts/1          2014-02-28 22:46  (:0.0)
```

当前的时间为:

2014年 02月 28日 星期五 22:54:15 CST

当前的文件的绝对路径为:

/home/ben/桌面

通过脚本5-3. sh的执行结果可以看出，倒引号能够使用Shell命令的执行结果代替在脚本中出现的Shell命令。

5.2 通配符和元字符

在Linux系统中，会使用一些特殊字符来完成普通字符无法完成的功能，如替换多个不需要特殊关系的字符，或使其具有特殊的功能。这类特殊字符一般被称为通配符和元字符。通配符用来匹配特定的字符序列，而元字符用来表示特殊的功能，这两种字符在Shell脚本中经常用到且很重要，需要读者认真学习。

5.2.1 使用通配符

在字符串的匹配以及查找过程中，有时需要查找的只是首字母或是中间的几个字符，而不去关心除此之外的其他字符。如果将所有可能出现的字符都一一列举显然不现实，此时就需要使用一种符号能够代替这些无关紧要的字符，从而完成整个字符串的匹配和查找过程，这类字符就是通配符。使用通配符可以匹配实际需要的任意字符串。在Shell中，经常使用的通配符如表5-1所示。

表5-1 常用通配符及其功能

通配符名称	作用	示例表达式
*	匹配任意个字符，可以为0个或多个	a*b
?	匹配一个字符	a?b
[list]	匹配list中的任意一个字符	a[abc]b
[! list]	匹配除list中的所有字符	a[! abc]b
[c1-c2]	匹配c1～c2中的任意字符，c1和c2之间必须为连续字符序列，如abc等	a[1-5]b
{string1,string2,...}	匹配string1或string2等字符串中的一个字符串	a{123, 456, 789}b

表5-1中列出了Shell中经常使用的通配符以及符号的作用和使用示例，下面分别讲述通配符的使用。

通配符“*”的作用是可以匹配任意个字符，包括0个或多个字符。a*b表示a和b之间可以有0个字符，也可以有任意个字符，如ab、a1b、a123b1gfewagfewfb等，都可以匹配a*b。也就是以字符a开头，而以字符b结尾的所有字符都能匹配a*b。通配符“*”的使用方法如示例脚本5-4.sh所示。

```
#示例脚本5-4. sh 通配符星号*的使用
#!/bin/bash
```

```
echo 使用通配符星号\"*\"  
mkdir file1 file2 fiLE3 FILE4 FILE5
```

```
echo 只使用1个字符f  
ls f*
```

```
echo 使用2个字符fi  
ls fi*
```

```
echo 使用3个字符fil  
ls fil*
```

```
echo 使用4个字符fiile  
ls 'file*' 
```

对示例脚本5-4. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-4. sh  
ben@ben-laptop:~ $ ./ 5-4. sh  
使用通配符星号\"*\"  
只使用1个字符f  
file1:  
  
file2:  
  
FILE3:
```

```
FILE4:
```

```
使用2个字符fi
```

```
file1:
```

```
file2:
```

```
FILE3:
```

```
使用3个字符fil
```

```
file1:
```

```
file2:
```

通配符“?”和通配符“*”使用方式类似，其区别就是通配符“?”只能匹配一个字符，而不是任意个字符，如示例表达式“a? b”表示以字符a开头，并且以字符b结尾，字符a和b之间只能存在一个字符，该字符可以为任意字符，如a1b、aab、avb等，都可以匹配“a?b”。但是ab、a123b因为字符a和b之间的字符个数不同而不能匹配“a? b”。通配符“?”的使用方法如示例脚本5-5. sh所示。

```
#示例脚本5-5. sh 通配符星号? 的使用
```

```
#!/bin/bash
```

```
echo 使用通配符星号\"?\\"
```

```
mkdir file file File File
```

```
echo 匹配第一个字符
```

```
ls ?ile
```

```
echo 匹配第后一个字符
```

```
ls fil?
```

```
echo 多个通配符共同使用
```

```
ls ?i*
```

对示例脚本5-5. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-5. sh
```

```
ben@ben-laptop:~ $ ./ 5-5. sh
```

使用通配符星号"?"

匹配第一个字符

File:

file:

匹配第后一个字符

file:

file:

多个通配符共同使用

File:

File:

```
file:
```

```
file:
```

通过示例脚本5-5. sh的执行结果可以看出，使用了通配符以后，在通配符位置的符号就被忽略掉，而是重点匹配其他的字符。用户可以根据实际需要来选择使用匹配单个字符的问号还是匹配零个或多个字符的星号。

在进行字符串的匹配过程中，还经常使用中括号[]来匹配确定的字符。中括号中的字符可以是ASCII码值连续的字符，也可以是任意的单个字符，还可以使中括号排除某些字符。关于这些匹配方式的使用如示例脚本5-6. sh所示。

```
#示例脚本5-6. sh 范围匹配的使用
#!/bin/bash

echo 使用中括号表示范围
mkdir hao123abc  hao145acd  hao124def    hao345cef    hao345ge
echo 显示"hao[1-3]*
ls | grep hao[1-3]*

echo 显示"hao[1-2][2-3][3-4]*
ls | grep hao[1-2][2-3][3-4]*
```

对示例脚本5-6. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-6. sh
```

```
ben@ben-laptop:~ $ ./ 5-6. sh
```

使用中括号表示范围

显示hao[1-3]*

```
hao123abc:
```

```
hao124def:
```

```
hao145acd:
```

```
hao345cef:
```

```
hao345gef:
```

显示hao[1-2][2-3][3-4]*

```
hao123abc:
```

```
hao124def:
```

通过示例脚本5-6. sh的执行结果可以看出，使用了范围匹配符号中括号以后，可以在一定的范围内匹配需要的字符，而在范围之内的字符，则会直接视为匹配失败。

5.2.2 使用元字符

元字符一般的定义就是作用于正则表达式的特殊字符，这些字符使

用特殊的格式使得里面出现的字符不再表示原来的含义，而具有特殊的意义，这方面类似于通配符。常用的元字符以及元字符的功能如表5-2所示。

表5-2 常用元字符及其功能

元字符名称	作用	使用示例
^	行首定位符	\^I\
\$	行尾定位符	\\$I\
.	匹配单个字符	\1.2\
*	匹配0个或多个位于“*”前的字符	\1*2\
[]	匹配一组字符中的任意一个	[list]
[x-y]	匹配指定范围内的任意字符	[a-g]
[^]	匹配不在指定字符组内的任意字符	[^abc]
\	用来转义元字符	*
\<	词首定位符	
\>	词尾定位符	
x\{m\}	字符x重复出现m次	x\{3\}
x\{m,\}	字符至少重复出现m次	x\{3, \}
x\{m,n\}	字符重复出现m 到 n 次	x\{3, 5\}

通过表5-2可以看到，元字符中有很多字符和通配符中类似，并且其作用也相同。但是通配符只是用来表示一些“无关紧要”的字符，而元字符则是可以用于正则表达式的特殊字符，这两种字符有着本质的差别。下面将一一介绍表5-2中出现的元字符。

符号“^”就是数字键6上对应的符号，用来匹配行首的字符，一般被称为脱字符。而美元符号“\$”则用来匹配行尾的字符，这些字符可以是单个字符，也可以是多个字符，可以根据实际需要灵活确定。如果需要匹配多个字符，需要使用引号括起来，从而组成一个字符集合。如果不使用引号，那么就只会匹配紧挨着脱字符或美元符号的字符，这两个符号的使用如示例脚本5-7. sh所示。

```
#示例脚本5-7. sh 元字符的使用
#!/bin/bash

mkdir hello123 Hello123 123hello 123HAO
echo "使用脱字符匹配开头字符"
echo 匹配开头字符是小写字符
ls | grep [^a-z]
echo

e c h o 显示开头字符是大写字符H
ls | grep [^H]

echo "使用美元符号匹配结尾字符"
echo 显示结尾字符是数字
ls | grep [1-9$]
```

```
echo
```

echo 显示结尾字符是大写字符

```
ls | grep [A-Z$]
```

对示例脚本5-7. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-7. sh
```

```
ben@ben-laptop:~ $ ./ 5-7. sh
```

使用脱字符匹配开头字符

匹配开头字符是小写字符

```
hello123
```

显示开头字符是大写字符H

```
Hello123
```

使用美元符号匹配结尾字符

显示结尾字符是数字

```
hello123
```

```
Hello123
```

显示结尾字符是大写字符

```
123HA0
```

通过示例脚本5-7. sh的执行结果可以看出，当使用脱字符和美元符号进行匹配时，可以只匹配开头字符和结尾字符，而不关心其他的字

符。

符号“*”（星号）、“.”（点号）、“[]”（中括号）分别用来匹配0个或多个字符、单个字符以及在某个范围内的字符，这3个符号的使用方式和通配符中的相同符号类似，使用方法可以参照上面的相关示例脚本。

使用元字符还可以用来确定某个字符出现的次数，如符号“x{m\}”表示字符x出现过m次；符号“x\{m,\}”则表示字符x至少重复出现m次，出现次数少于m的字符串均视为无效字符串；符号“x\{m,n\}”表示字符x的出现次数为m到n次之间，这样可以更加精确地获取字符的出现次数。除了确定字符的出现次数之外，还可以确定字符串的出现次数，只要将字符换为字符串即可。关于这3个符号的使用如示例脚本5-8. sh所示。

```
#示例脚本5-8. sh 匹配出现频率
#!/bin/bash

mkdir hello Hello HELLO HEllO HELLo helloHELlo
echo "匹配字符l至少出现2次"
ls | grep l\{2\}                                #至少出现2次l
echo

echo "匹配字符l出现1~2次"
ls  | grep l\{1,2\}"                            #匹配字符l出现1~2次
```

对示例脚本5-8. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-8.sh
```

```
ben@ben-laptop:~ $ ./ 5-8.sh
```

匹配字符l至少出现2次

```
helloHELlo
```

匹配字符l出现1~2次

```
hello
```

```
Hello
```

```
Hello
```

```
HELlo
```

通过示例脚本5-8.sh的执行结果可以看出，如果需要判断某个字符出现的频率，可以使用大括号来进行判断，符合要求的才会显示出来。

注意

在使用时，反斜线加字符的字符序列称为元字符，单独使用时某些字符为通配符，二者不是相同的概念。

5.3 管道

在使用Shell的过程中，有时会遇到两个或多个命令依次执行而上一个命令的执行结果作为当前命令的参数，如果使用临时变量存储上一个命令的执行结果而作为后面的命令的参数，这样显然是不容易实现的，

因为命令的执行结果多种多样，并不是每一个命令的执行结果都能使用变量存储。这样就需要使用一种新的符号来实现这种功能，这种符号就是管道。

在编写Shell脚本乃至执行Shell命令时，管道的使用都非常频繁。从根本上来说，管道是一种进程间的通信机制，同时也是一种文件（在Linux系统中，所有的内容都被视为文件）。因此可以使用操作文件的方式对管道进行读写操作。在进行读取管道操作时，其存储空间只有4KB，而数据一旦被读取以后，管道将被清空，其中的数据也将被抛弃，从而释放空间用来存储更多的数据。

在Shell脚本中，经常使用管道符来连接两个进程，即使用一个Shell命令去处理另外一个命令的执行结果，从而获取新的结果。这两个命令能够使前一个命令运行完毕后后一个命令自动运行。管道使用符号竖线“|”表示，竖线前面的内容的执行结果将作为后面命令的执行参数。

如在查看示例脚本5-9. sh的部分内容时，可以将cat命令的显示结果作为head命令的输入参数，当head命令执行完毕后，文件 5-9. sh的前3行就显示在计算机的屏幕上。

```
#示例脚本5-9 管道符号的使用
#!/bin/bash

echo 使用管道符查看文件内容
cat 5-9. sh | head -n 3
```

对示例脚本5-9. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-9.sh  
ben@ben-laptop:~ $ ./ 5-9.sh  
使用管道符查看文件内容  
#!/bin/bash  
  
echo 使用管道符查看文件内容
```

通过示例脚本5-9.sh的执行结果可以看出，在使用了管道符以后，可以实现多个命令对同一个输入内容进行处理，从而得到最终想要的数据。

在实际的脚本编写过程中，还可以使多个管道符号同时出现在一个语句中，这样使得第一个命令需要后一个命令的执行结果作为输入的参数，而第二个命令同样需要其后面命令的执行结果作为输入参数，以此类推，直至最后一个命令被执行，这样才具有了所有的命令的执行条件，而该语句的最终输出结果为第一个命令的执行结果。使用这种方式能够实现一条语句中所有命令的自动执行，从而当最后一个命令运行成功之后，得到最终的结果。

注意

从管道读数据是一次性操作，数据一旦被读取，它就从管道中被抛弃，释放空间以便写入更多的数据。

5.4 其他特殊字符介绍

在Linux系统中编写Shell脚本时，还有其他的特殊字符出现，如后台运行符、括号、分号等。这几个字符也经常使用，本小节将重点介绍括号和分号的用法，而对于后台运行符，在后面的章节中将详细介绍。

5.4.1 后台运行符

在操作Linux系统的过程中，有一些程序，如监听程序等，在这些程序执行时，用户还可以执行其他的操作，而不影响程序的运行。因此为了更加合理地利用计算机资源，可以将这些不需要在前台执行的程序放到计算机后台去执行，这样既保证了程序的正常运行，又可以使前台的操作能够继续进行。

在执行Shell脚本或Shell命令时，同样可以实现后台执行。在确定要使用后台执行时，只需要在执行Shell脚本或Shell命令时，在后面添加后台运行符“&”，即可实现脚本或命令的后台运行。

后台运行符的使用使得在执行了脚本以后，还可以进行其他的操作，而不用等着脚本执行完之后再进行其他的操作。如编写可以不断向一个普通文件中写入字符的脚本5-10. sh，当使用普通的运行方式时，当前的Shell终端中无法再进行其他的操作。而使用了后台运行的方式以后，仍然可以运行其他的命令，同时使用后台运行的方式使其运行。示例脚本5-10. sh的内容如下：

```
#示例脚本5-10. sh 后台运行符号的使用
#!/bin/bash
```

```
echo 后台运行符的简单使用
```

```
sleep 100
```

对示例脚本5-10. sh执行权限后，不使用后台运行符号，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-10. sh
```

```
ben@ben-laptop:~ $ ./ 5-10. sh
```

```
后台运行符的简单使用
```

```
^C
```

在使用了后台运行符号以后，执行结果如下：

```
ben@ben-laptop:~ $ ./ 5-10. sh &
```

```
[1] 2744
```

使用ps命令查看在进程列表中是否存在进程5-10. sh的运行，方法如下：

```
/实例脚本$ ps -ef | grep 5-10* | grep -v grep
```

```
ben      2744 2016 0 10:43 pts/0 00:00:00 /bin/bash ./.sh
```

通过这两次的执行结果可以看出，在不使用后台运行符时，只能等待脚本运行结束或者直接中断脚本的运行，当前的Shell终端中才可以进行各种操作。而使用了后台运行符以后，脚本执行的进程存在于系统中，并且可以在终端中进行其他的操作，而不会受到任何的影响。

如果要结束后台运行的程序，可以先查询该程序的进程号，然后使用kill命令杀掉该进程，即可结束进程，如图5-1所示。

```
ben@ben-laptop:~$ ps -ef | grep 5-10 | grep -v grep
ben      2089  2073  0 18:17 pts/1    00:00:00 /bin/bash ./5-10.sh
ben@ben-laptop:~$ kill 2089
ben@ben-laptop:~$ ps -ef | grep 5-10 | grep -v grep
ben@ben-laptop:~$
```

图5-1 终结后台运行程序

通过图5-1中的操作可以看出，在使用了后台运行符以后，正在运行的进程可以通过ps命令查到，而通过grep命令来获取该命令的详细信息以后，使用kill命令可以将该后台运行的进程删除。

注意

grep命令主要用于内容的匹配，从而获取需要的内容，该命令的使用方法将在后面的章节中进行介绍。

5.4.2 括号

Shell脚本中经常使用的括号包括大括号“{}”和小括号“()”，这两种括号在使用时既有相同点，又存在一定的不同，下面将分别讲述这两种括号的使用。

大括号“{}”可用于变量的分辨。如在使用变量时可以使用美元符号“\$”加变量名的方式获取变量的具体值，如下面的使用方式所示：

```
$string = "hello world"  
$num = 10
```

但是如果在变量名后面紧跟着其他的字符时，就需要使用大括号区分哪些是代表变量名的特殊字符，哪些是普通字符，从而避免出现歧义，如示例脚本5-11. sh所示。

```
#示例脚本5-11. sh 括号的使用  
#! /bin/bash  
  
echo “请输入登录用户的姓名:”  
read name  
echo 不使用括号  
echo 欢迎$name的登录  
echo 使用括号  
echo欢迎${name}的登录
```

对示例脚本5-11. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-11. sh  
ben@ben-laptop:~ $ ./ 5-11. sh  
请输入登录用户的姓名：  
ben  
不使用括号  
欢迎ben的登录  
使用括号
```

欢迎ben的登录

通过示例脚本5-11. sh的执行结果可以看到，如果在变量名后面添加其他的字符，Shell会将符号“\$”后面所有的字符都视为变量名，从而引起无法正常地使用变量。解决这个问题的方式就是使用大括号“{}”将变量名括起来，从而获取正确的变量名。

小括号“（）”一般用于命令的替换，此时小括号等同于倒引号。当Shell脚本在执行时，如果发现小括号（或倒引号），那么首先将括号（或倒引号）里面的命令执行，并将执行结果放到命令出现的地方，从而实现命令的替换，如示例脚本5-12. sh所示。

```
#示例脚本5-12. sh 小括号的使用
#!/bin/bash

echo 使用小括号
echo "当前的登录用户为"${who}
echo 使用倒引号
echo "当前的登录用户为``who`"
```

对示例脚本5-12. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-12. sh
ben@ben-laptop:~ $ ./ 5-12. sh
使用小括号
"当前的登录用户为"ben  tty7 2014-02-28 22:42 (:0)  ben pts/0 201
使用倒引号
```

```
"当前的登录用户为"ben  tty7 2014-02-28 22:42 (:0) ben pts/0 201
```

通过示例脚本5-12. sh的执行结果可以看出，小括号和倒引号都能够使用命令的执行结果替换命令本身。

小括号除了用于命令替换以外，还可以用于命令组以及数组的初始化，这些内容将在后面的章节中进行讲述。

注意

小括号可以两对括号一同使用，和test命令一起用来判断逻辑表达式是否成立，这些内容将在讲述test命令时详细介绍。

5.4.3 分号

分号在Shell脚本中的作用是分隔语句。一行中一般只存在一条语句，即一个Shell命令以及命令所需要的参数所组成语句，此时在语句的末尾不需要任何符号，可以直接在下一行编写其他语句。但是如果在一行中出现多个语句，即多个单独的命令在一行中存在，此时需要在命令的结尾添加分号，从而区分每一个命令。这样就可以使得每一个命令都能够正确无误地被执行，如示例脚本5-13. sh所示

```
#示例脚本5-13. sh 分号的使用
#!/bin/bash
```

```
echo 使用分号依次显示环境变量中的用户主目录、使用的Shell类型
echo $HOME;echo $SHELL

echo 不使用分号依次显示环境变量中的用户主目录、使用的Shell类型
echo $HOME
echo $SHELL
echo $HOMEEcho $SHELL
```

对示例脚本5-13. sh执行权限后，执行结果如下：

```
ben@ben-laptop:~ $chmod u+x 5-13. sh
ben@ben-laptop:~ $ ./ 5-13. sh
使用分号依次显示环境变量中的用户主目录、使用的Shell类型
/home/ben
/bin/bash
不使用分号依次显示环境变量中的用户主目录、使用的Shell类型
/home/ben
/bin/bash
/bin/bash
```

通过示例脚本5-13. sh的执行结果可以看出，使用分号可以使在一行中出现多条需要执行的语句，而不使用分号，为了使得命令的执行更加正确，每一行中只出现一条命令，这样可防止前面的命令在执行时，将后面的命令作为参数处理。

5.5 小结

本章主要讲述了Shell脚本中经常出现的特殊符号，即引号、通配符和元字符、管道以及其他特殊的符号（如后台运行符、括号、分号等符号）的用法，准确地理解这些符号的使用方式可以更加准确地了解脚本是如何执行的，对编写出高质量的Shell脚本也有非常大的帮助。

在Shell脚本中，引号一般包括3种：单引号、双引号、倒引号。单引号的作用是将引号中的内容全部作为普通字符处理，在单引号中没有任何的特殊字符，而双引号除了将符号\$、倒引号(`)和反斜线(\)作为特殊字符处理之外，其他的字符仍然作为普通的字符处理。而倒引号则是将字符串中的命令执行结果来替换作为命令出现在字符串。这3个符号可以使字符串的内容输出更加“丰富多彩”。

通配符和元字符都是Shell脚本中的特殊字符，在Shell脚本中具有特殊的功能。通配符和元字符一般用于字符串的查找和正则表达式等场合，通配符可以用来匹配任意不需要关心的字符，而元字符将匹配的字符的位置以及范围进行简单设定。通配符和元字符虽然使用方式类似，但是二者是不同的概念，希望读者不要混淆。

在Shell脚本中，管道的作用是将前一个命令的输出作为后一个命令的输入参数，从而将多个命令嵌套在一起，而最终的结果只是最后一个命令的执行结果，但是该命令的输入则是前一个命令的输出结果，以此类推，最终执行子一个命令。

在Shell脚本中，还有一些使用率不是很高的特殊符号，例如使程序在后台运行的后台运行符“&”、能够在一行中出现多个命令的分号，以及命令替换的小括号和大括号，这些符号虽然不经常使用，但是可以使编写的脚本更加灵活。

第6章 Linux中的文件处理

在第1章中已经介绍过，Linux系统的特点之一就是将所有的内容都作为文件进行处理，包括所有的硬件设备（如硬盘、网卡、显卡等）、普通文件（普通的文本文件等）等。因此对文件的处理是Linux系统中非常重要的部分，本章将详细介绍在Linux系统中文件的处理以及常用的文件类操作。

本章的主要知识如下：

- Linux系统中的文件类型，设备文件、连接文件以及文件描述符
- 特殊的文件，标准输入、标准输出、标准错误
- 重定向，包括输入重定向、输出重定向以及临时重定向和永久重定向
- 标准输出和标准错误的基本方式

6.1 Linux中的文件类型

在一般的Linux文件系统中，所有的内容，包括硬件设备等都是按照文件来处理的，而在Linux系统中，文件的主要类型如表6-1所示。

表6-1 Linux系统中文件的分类

文件类型	文件组成	文件作用
文本文件	主要是ASCII字符	记录普通文件，如源文

		件、txt文件等
二进制文件	由二进制码0、1组成	由计算机可直接执行的文件
目录文件	存放着文件名和文件索引结点之间的关联关系	类似于Windows系统中的文件夹，用来存放其他各种类型的文件
链接文件	指向其他的文件，类似于快捷方式	执行链接文件时，能够指向最终的执行目标
设备文件	存储设备信息	表示硬件设备
管道文件	进程间通信的内容	进程间的通信

查看某个文件属于哪种文件，可以使用ls命令以及-l选项。在显示文件的详细信息时，根据文件属性的第一个字符来判断文件的属性，还可以使用命令file来为文件“验明正身”。关于这两个命令的使用方式在第3章中已经做了介绍，对此不是很熟悉的读者可以参考第3章的相关内容。

6.1.1 设备文件

在Linux系统中，所有的外部设备都被看作文件来管理，而不同的设备对应着不同的设备文件。用户在使用外设时就像使用普通文件一样方便，可以调用相应的函数或指令对设备进行各种操作。

在Linux系统中，设备文件存放在/dev目录下，它使用设备的主设备号和次设备号来区分指定的外设。其中，主设备号用来说明设备类型，

即该设备属于什么样的设备，而次设备号说明具体指哪一个设备。如对于USB接口设备来说，主设备号用来表示该设备属于USB设备，而次设备号则用来表示该设备具体属于哪一个USB设备。

在Linux系统中，设备文件主要是块设备文件和字符设备文件。块设备的主要特点是可以随机读写，而最常见的块设备就是磁盘。在系统中常用文件/dev/hda1、/dev/sda2和/dev/fd0等来表示。在Linux的分区表示中，硬盘用hd来表示，而第1块硬盘为hda，第2块为hdb，以此类推。而hda1则表示第1块硬盘的第1个分区，而hda2则表示第1块硬盘的第2个分区，以此类推。而在Linux中，sda也用来表示硬盘，其表示方式也和hda的表示方式相同，不同的是sda表示SCSI硬盘，而had则表示IDE接口类的硬盘。

在Linux系统中，常见的字符设备是打印机和终端，字符设备可以接受字符流，也可以输出字符流。对于字符设备来说，/dev/null是一个非常有用的字符设备文件，一般将/dev/null称为空设备。如果将程序的输出结果重定向到/dev/null，则看不到任何输出信息。/dev/null等价于一个只写文件，所有写入它的内容都会被自动清除，而尝试读取文件中的内容则什么也读不到，然而文件/dev/null对命令行和脚本都非常有用，可以用来清除某些不需要的内容。空设备文件/dev/null的使用方式如示例脚本6-1. sh所示。

```
#示例脚本6-1. sh空文件/dev/null的使用
```

```
#! /bin/bash
```

```
echo 查看空文件/dev/null的内容
```

```
cat /dev/null
```

```
echo  
  
echo 向文件中写入数据  
echo 6-1. sh >/dev/null  
  
echo 写入数据后，查看文件的内容  
cat /dev/null
```

为示例脚本6-1. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~# chmod u+x 6-1. sh  
ben@ben-laptop:~# ./6-1. sh  
查看空文件/dev/null的内容  
  
向文件中写入数据  
写入数据后，查看文件的内容
```

通过脚本6-1. sh执行结果可以看出，文件/dev/null类似于回收站，存入该文件中的任何内容都将被直接删除，而不会进行任何的存储。可以使用该文件来实现对没有必要存储的内容的删除功能。

6.1.2 设备文件的挂载和卸载

如果设备文件本身不是系统自带的设备，如外接U盘、光驱等，在使用之前应该首先对设备进行挂载，使得该设备在Linux系统中按照文

件的方式出现，然后才能使用。挂载设备一般使用mount命令，该命令的一般格式如下：

```
mount [选项] 设备名 目标目录
```

mount命令需要首先指定需要挂载的设备名，然后要确定挂载到哪个目录中，除此之外，还可以根据需要添加适当的选项。mount命令的常用选项如表6-2所示。

表6-2 mount命令的常用选项

选项	作用	备注
-t	指定系统文件类型	可以不指定，操作系统会自动分辨设备的文件系统类型
-o loop	用来把一个文件当成硬盘分区挂接上系统	需要使用-o选项
-o ro	采用只读方式挂接设备	需要使用-o选项
-o rw	采用读写方式挂接设备	需要使用-o选项
-o iocharset	指定访问文件系统所用字符集	需要使用-o选项

操作系统在设备挂载时会自动对设备的文件系统进行分析，从而得到最终的结果。这些文件系统的类型，虽然不需要用户自己指定，但还是需要了解的。常用的文件系统类型如表6-3所示。

表6-3 常用的文件系统类型

文件系统类型	表示符
DOS fat16文件系统	msdos
Windows 9x fat32文件系统	vfat
Windows NT ntfs文件系统	ntfs
Windows文件网络共享	smbfs
Unix（Linux）文件网络共享	nfs

使用mount命令可以挂载多种设备类型，在本小节中，以最常用的挂载U盘为例子讲解如何使用mount命令进行挂载。挂载设备一般分为3步，如图6-1所示。

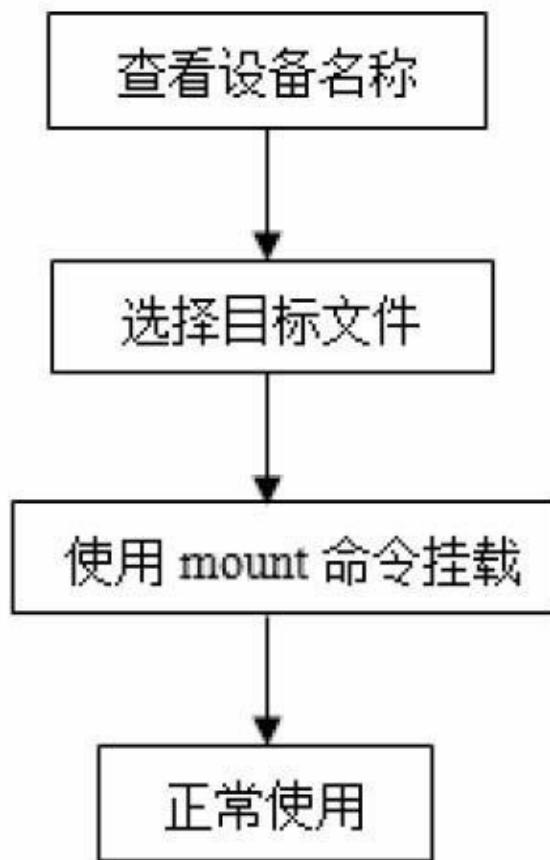


图6-1 设备挂载步骤

挂载设备时，首先要明确需要挂载的设备的名称。这个名称可以使用fdisk命令查看。硬盘的信息也可以使用fdisk命令来查看，fdisk命令加上-l选项可以显示当前所有计算机上所有的硬盘分区。如以下操作所示：

```
ben@ben-laptop:~$ sudo fdisk -l

Disk /dev/sda: 640.1 GB, 640135028736 bytes
255 heads, 63 sectors/track, 77825 cylinders
```

```

Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical) : 512 bytes / 512 bytes
I/O size (minimum/optimal) : 512 bytes / 512 bytes
Disk identifier: 0x1fbebe617

```

Device	Boot	Start	End	Blocks	Id	Sy
/dev/sda1	*	1	10444	83891398+	7	HFS+
/dev/sda2		10445	77825	541237882+	f	W95
/dev/sda5		10445	27285	135275301	7	HFS+
/dev/sda6		27286	44126	135275301	7	HFS+
/dev/sda7		44127	60967	135275301	7	HFS+
/dev/sda8		60968	77825	135411853+	7	HFS+

```

Disk /dev/sdb: 320.1 GB, 320072932864 bytes
255 heads, 63 sectors/track, 38913 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical) : 512 bytes / 512 bytes
I/O size (minimum/optimal) : 512 bytes / 512 bytes
Disk identifier: 0x552c7caa

```

Device	Boot	Start	End	Blocks	Id	Sy
/dev/sdb1	*	1	6528	52436128+	7	HFS+
/dev/sdb2		6529	26675	161827617+	f	W95
/dev/sdb3		26675	26924	1999872	82	Linux
/dev/sdb4		26924	38914	96306176	83	Linux
/dev/sdb5		6529	16973	83899431	7	HFS+
/dev/sdb6		16974	21448	35943424	7	HFS+

使用fdisk命令显示出设备名称之后，可以根据设备名称进行设备的挂载。挂载实际上就是将设备中的内容在计算机中显示出来。显示的第一步就是创建目标文件。目标文件必须是目录文件，这样可以将设备文件中的内容放置到目标文件中。目标文件可以使用mkdir命令进行创建。对于Linux操作系统来说，默认的目标目录是/media目录。在这个目录中，包含着当前已经挂载的设备。如以下操作所示：

```
ben@ben-laptop:~$ ls -l /media
总用量 56
drwx——— 1 ben  ben   8192 2014-01-29 02:05 0003D2FD000310
drwx——— 1 ben  ben  24576 2014-01-29 02:05 E8F44289F44259
drwx——— 2 root  root  4096 2014-04-05 22:16 工作
drwx——— 1 ben  ben   4096 2014-05-20 10:41 工作_
drwx——— 1 ben  ben   8192 2014-07-05 22:08 工作_
drwx——— 1 ben  ben   8192 2014-06-07 15:31 娱乐
ben@ben-laptop:~$
```

在上面的操作中，显示了笔者计算机中当前挂载的设备，这些设备都以文件的形式存在于系统中。通过文件的详细信息也可以看出，这些文件都是以目录文件的方式存在于系统中的。目标文件可以创建在/media目录中，也可以创建在其他的目录中，可以根据实际需要进行创建。创建目标目录时只需要使用mkdir命令创建一个新的文件即可。

在选择好目标目录以后，就可以使用mount命令来进行设备的挂载了。挂载过程如示例脚本6-2. sh所示。

```
#示例脚本6-2. sh 设备的挂载
#!/bin/bash

echo 使用mount命令挂载硬盘/dev/sda7到目录 /home/media
sudo mount /dev/sda7 /home/media

echo 挂载结束
echo 查看挂载后的目录文件
ls -l /home/media
```

为示例脚本6-2. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~/# chmod u+x 6-2.sh
ben@ben-laptop:~/# ./6-2.sh
使用mount命令挂载硬盘/dev/sda7到目录 /home/media
挂载结束
查看挂载后的目录文件
总用量 389
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-1.sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-2.sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-3.sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-4.sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-5.sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-6.sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-7.sh
```

当挂载设备成功之后，在目标目录中就会出现设备中的所有文件。而对设备进行操作时，就变成了对这个文件中的内容进行操作。

注意

在进行设备的挂载时，需要使用root用户来执行mount命令，对于普通用户不允许进行设备的挂载。

如果选择的目标目录中存在其他内容时，挂载以后，该文件中的内容就不再显示，而只有当挂载到该目录中的设备卸载以后，才会正常显示原来的内容，如示例脚本6-3. sh所示。

```
#示例脚本6-3. sh 挂载后目录中内容的变化
#!/bin/bash

echo 查看目标目录 /home/media中的内容
echo 设备挂载
sudo mount /dev/sda7 /home/media

echo 挂载结束
echo 查看挂载后的目录文件
ls -l /home/media
```

为示例脚本6-3. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~# chmod u+x 6-3. sh
```

```
ben@ben-laptop:~/# ./6-3. sh
查看目标目录 /home/media中的内容
设备挂载
挂载结束
查看挂载后的目录文件
总用量 389

-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-1. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-2. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-3. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-4. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-5. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-6. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-7. sh
```

当设备不再使用时，即使是能支持热插热拔的设备，也最好进行设备的卸载。卸载时使用命令umount来进行，该命令的使用格式如下：

umount 目标目录或设备名

在使用umount命令卸载设备时，一般只需要添加目标路或设备名即可，而不需要其他的参数。umount命令的使用方式如示例脚本6-4. sh所示。

```
#示例脚本6-4. sh 设备的卸载
#!/bin/bash
```

```
echo 查看挂载后的目录文件
```

```
ls -l /home/media
```

```
echo 设备卸载
```

```
sudo umount /dev/sda7 /home/media
```

```
echo 卸载结束
```

```
echo 查看卸载后的目录文件
```

```
ls -l /home/media
```

为示例脚本6-4. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~# chmod u+x 6-4. sh
```

```
ben@ben-laptop:~# ./6-4. sh
```

```
查看挂载后的目录文件
```

```
总用量 389
```

```
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-1. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-2. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-3. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-4. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-5. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-6. sh
-rwxrwxrwx 1 root root 5224 2013-10-03 11:03 6-7. sh
```

```
设备卸载
```

```
卸载结束
```

查看卸载后的目录文件

通过示例脚本6-4. sh中的设备卸载过程可以看出，使用umount命令卸载以后，原本挂载到目录/home/media中的内容将不再显示，该设备也就不能再使用。

注意

umount命令和mount命令一样，也需要使用root用户权限，对于普通用户不允许进行设备的卸载。

6.1.3 链接文件

在Windows系统中，经常会遇到一些快捷方式。这些快捷方式是Windows系统为了方便用户的使用而提供的一种能够实现快速启动程序及打开文件或文件夹的方法。它是应用程序的快速连接。快捷方式类似于一个遥控器，通过按下遥控器上的按键来实现电视节目的选择。对于用户来说，虽然操作的是按键，但是实际上操作的却是电视。

对于Linux系统来说，链接文件也是文件的一种，它的作用就是实现文件的链接。当打开某个当前的链接文件时，即可打开该链接文件指向的目标文件。

在Linux系统中，链接文件分为软链接和硬链接。软连接就是Windows系统中的快捷方式，其作用仅为文件指向的作用，而不是具体

的文件。当目标文件发生任何变化时，软链接都不会发生任何变化。

对于硬链接来说，链接文件和指向的目标文件是相同的，类似于目标文件的备份。链接文件和目标文件是同步的，如果目标文件发生了任何的变化，链接文件都会随之变化。

注意

硬链接不占用硬盘资源，并且硬链接只能作用于文件，但是不能“跨越磁盘分区作业”。

硬链接和软连接虽然都是链接文件，但是二者还是有很大的区别的，是主要的区别在于以下几点。

- 硬链接是创建一个指向文件的节点（inode），这样可以有效地防止文件被误删除。因为在Linux系统中，删除一个文件的实质是删除该文件的节点（inode）信息，从而切断了节点（inode）与文件之间的联系。当给一个文件创建了硬链接之后，删除文件时，只会减少文件的链接数（即节点数），当节点数为0时，才会彻底删除该文件。软链接类似于Windows系统中的快捷方式，是一个完整的文件，链接文件和目标文件之间只有指向的关系。当原文件删除后，链接文件指向的内容就没有了，因此这个软链接就失效了。
- 文件的大小不一样。由于硬链接指向目的文件，所以硬链接的文件大小和原文件大小是一致的。而软链接是新的文件，文件中的内容是目的文件的文件名，所以软链接文件的一般是

4KB，即一个节点的大小。

- 硬链接不能跨目录、跨分区做链接。而软链接不但可以实现跨目录链接，还可以实现跨分区链接。
- 创建硬链接时使用命令ln，而不需要使用任何选项。而创建软链接则需要添加-s选项，如示例脚本6-5. sh所示。

```
#示例脚本6-5. sh链接文件的使用

#!/bin/bash

mv 6-2. sh test.sh

ln test.sh HLtest #file为原文件名, filenew为新文件名。#创建硬软链接
ls -l #显示文件属性

ln -s test.sh SLtest #file为原文件名, filenew为新文件名 #创建软链接
ls -l #显示文件属性

rm test.sh

cat HLtest SLtest
```

为示例脚本6-5. sh赋予可执行权限后执行脚本，结果如下：

```
root@ben-laptop:~/# chmod u+x 6-5. sh
root@ben-laptop:~/# ./6-5. sh
创建硬链接命令
-rwxrwxrwx 1 ben ben 345 2014-04-07 22:57 HLtest
-rwxrwxrwx 1 ben ben 441 2014-04-07 23:00 test.sh
创建软链接命令
```

```
lrwxrwxrwx 1 ben ben 22 2014-04-07 22:57 SLtest ->test.sh
-rwxrwxrwx 1 ben ben 441 2014-04-07 23:00 test.sh
删除test.sh
查看链接文件SLtest
cat: SLtest: 没有那个文件或目录
查看链接文件HLtest
#!/bin/bash

mv 6-2.sh test.sh
ln test.sh HLtest #file为原文件名, filenew为新文件名。#创建硬软链接
—More—
```

通过示例脚本的执行结果可以看出，不管是硬链接还是软连接，文件的属性都是链接文件，表示文件属性的字符为“l”。当删除了原文件以后，在查看软链接时，会提示“文件不存在”，而对于硬链接文件来说，文件仍然存在并且文件中的内容和原文件的内容完全一样。当删除了所有的硬链接文件以后，再查看该文件时，才会提示“文件不存在”的错误。SLtest链接文件的属性如图6-2所示。



图6-2 SLtest链接文件属性

通过图6-2可以看出，当软链接指向的文件被删除后，链接文件的属性中显示了链接已断开的状态，从而进一步表明改链接文件已经不能正常使用。

6.1.4 文件描述符

在Linux系统中，所有的设备都被当做文件，除了用节点（inode）来描述文件以外，还可以使用文件描述符来表示文件。文件描述符在形式上是一个非负整数，即从0开始计算，每打开一个文件，文件描述符就自动加1。实际上，文件描述符是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核就会返回一个文件描述符。

文件描述符的有效范围是从0到OPEN_MAX。一般来说，每个进程最多可以打开64个文件，文件描述符的范围就是从0到63。文件描述符是由无符号整数表示的句柄，进程使用它来标识打开的文件。文件描述符与文件的相关信息（如文件的打开方式、文件的位置、文件的类型等）相关联，这些信息被称作文件的上下文。在Linux系统中编写程序时，可以通过相应的结构体来查询这些信息。

注意

文件描述符一般从3开始私用，前3个文件描述符分别对应着标准输入、标准输出与标准错误。

6.2 标准输入、输出与错误

在执行一个Shell命令行时通常会自动打开3个标准文件。标准输入文件（STDIN），通常对应终端的键盘；标准输出文件（STDOUT）和标准错误输出文件（STDERR）都对应终端的屏幕。进程将从标准输入文件中得到输入数据，将正常输出数据输出到标准输出文件，而将错误信息送到标准错误文件中。

6.2.1 标准输入

对于计算机来说，在进行输入操作时，输入信息的来源可以是内存、文件、数据库等，而标准输入就是从键盘上获取输入信息，也就是从键盘上键入字符来作为输入信息，这种输入一般被认为是标准输入。在Linux系统中，使用STDIN来表示，对应的设备文件是/dev/stdin，而在操作时，需要从键盘上输入字符供程序使用。

注意

在Shell脚本中，标准输入对应的文件描述符为0。

6.2.2 标准输出和标准错误

对于计算机来说，系统输出的信息可以存放在内存、文件、数据库

中，而标准输出就是将需要输出的信息显示到屏幕上。在Linux系统中，使用STDOUT来表示，对应的设备文件是/dev/stdout，在使用标准输出时，所有的信息都会显示在屏幕上。

注意

在Shell脚本中，标准输出对应的文件描述符为1。

标准错误的使用方式和标准输出类似，都需要将信息发送到屏幕上显示。对于标准错误来说，显示的信息是错误信息。而在Linux系统中，一般使用STDERR来表示标准错误。

注意

在Shell脚本中，标准错误对应的文件描述符为2。

6.3 重定向

对于计算机中的输入输出操作来说，除了标准输入、标准输出以外，还存在其他的输入输出途径，如当数据量很大时，可以选择从文件中获取输入信息，而为了更好地保存、备份数据，可以将输出到屏幕上的信息转存到文件、数据库中，这就将输入输出的方向进行改变，这种改变就是重定向。在Linux系统中，最常用的重定向包括输入重定向和输出重定向，本小节就将重点介绍这两方面的内容。

6.3.1 重定向的定义

重定向一般来说是IO重定向，即输入输出的重定向。简单来说就是捕捉来自非键盘输入的数据内容作为输入信息，然后把本来应该输出到显示器屏幕上的内容输出到其他的地方。重定向就是将标准的输入（来自键盘的输入）或者标准的输出（输出到屏幕）更改成其他的方式，如示例脚本6-6. sh所示：

```
#示例脚本6-6. sh重定向符的简单使用
#!/bin/bash

echo '使用ls命令显示目录/home中部分的内容'
ls -l /home

echo 'home重定向ls命令显示的目录/home中部分的内容'
ls -l /home 1>home.txt

echo 显示重定向文件home.txt中的内容
cat home.txt
```

为示例脚本6-6. sh赋予可执行权限后执行脚本，结果如下：

```
root@ben-laptop:~# chmod u+x 6-6. sh
root@ben-laptop:~# ./6-6. sh
使用ls命令显示目录/home中部分的内容
总用量 8
```

```
drwxr-xr-x 52 ben  ben  4096 2014-04-07 23:07 ben
drwxr-xr-x  2 root root 4096 2014-04-05 22:08 media
home重定向ls命令显示的目录/home中部分的内容
显示重定向文件home.txt中的内容
总用量 8
drwxr-xr-x 52 ben  ben  4096 2014-04-07 23:07 ben
drwxr-xr-x  2 root root 4096 2014-04-05 22:08 media
```

上面的命令显示了重定向的操作方式。第一条命令显示目录文件home中的所有内容到屏幕上，这种输出操作属于标准输出。而第二句使用的方式是将ls命令的执行结果保存在home.txt文件中，而不再输出到屏幕上，这就是将标准输出重定向到文件dev.txt中。使用重定向以后，原本应该显示在屏幕上的内容转到了文件home.txt中，这就是简单的输出重定向。

注意

对于标准输出来说，系统默认的文件描述符就是1，因此1可以省略，而直接使用类似于ls /dev >dev.txt的形式。

6.3.2 输入重定向

如果某个脚本在运行时需要进行大量的输入操作，这个输入过程不但需要花费很长的时间，还会造成数据的输入错误。解决这个问题的方式就是提前将需要输入的内容放到文件中，而在脚本执行时，将输入源

重定向到该文件，这样就减少了因为输入大量的数据而造成的时间浪费，还可以保证脚本获取到的数据的正确性和一致性。输入重定向是指把标准输入重定向到指定的文件中。也就是说，在重定向以后，输入可以不需要来自键盘的输入，而是来自一个指定的文件。所以说，输入重定向主要用于改变一个命令的输入源，特别是改变那些需要大量输入的输入源。

在进行输入重定向时，使用输入重定向符号“<”来表示，使用该符号的一般形式如下：

```
命令< 文件名
```

该符号类似于比较运算中的小于号，符号的左边是需要执行的命令，符号的右边是重定向后的数据输入源，可以用来提供左边的命令执行时需要的数据。如果在输入重定向符号的右边使用0，那么该输入还是使用标准输入作为命令的输入数据源。

示例脚本6-7. sh使用wc命令来获取输入信息中所包含的行数、单词数和字符数等信息。在该脚本中，使用输入重定向来统计文件中的单词数和字符数，以此来介绍输入重定向的作用和用法。

```
#示例脚本6-7. sh 输入重定向符
#!/bin/bash

echo 显示data.txt中的内容
cat data.txt
echo
```

```
echo 显示data.txt中的行数
```

```
wc -l< data.txt
```

```
echo 显示data.txt中的字符数
```

```
wc -c< data.txt
```

为示例脚本6-7. sh赋予可执行权限后执行脚本，结果如下：

```
root@ben-laptop:~# chmod u+x 6-7. sh
```

```
root@ben-laptop:~# ./6-7. sh
```

显示data.txt中的内容

1

h

1

显示data.txt中的行数

3

显示data.txt中的字符数

6

通过示例脚本6-7. sh的执行结果可以看出，在使用了输入重定向以后，执行wc命令时，需要统计的内容变成了文件data.txt中的内容，这样既减少了输入时系统的等待时间，也避免了在输入的过程中因为输入错误而产生脚本执行后统计结果不一致的情况。

在Linux系统中，由于大多数命令都以参数的形式在命令行上指定输入文件的文件名，所以输入重定向并不经常使用。尽管如此，当要使

用一个不接受文件名作为输入参数的命令，而需要的输入内容又存在于一个文件里时，就能用输入重定向解决问题。

6.3.3 输出重定向

在Shell终端执行命令时，有些命令的执行结果需要验证或保存，但是其输出的长度却非常长，而无法在屏幕中一次显示出来，这样，就无法获取执行结果中的特定内容。这种情况下，就需要把输出的内容重新定位到某个特定的文件中进行保存，然后通过读取文件来获取需要的数据，这就是输出重定向。

输出重定向是指把命令（或可执行程序）的标准输出或标准错误输出重新定向到指定文件中。这样，该命令的输出就不会显示在屏幕上，而是写入到指定文件中。输出重定向除了能够将标准输入定向到文件中以外，还可以把一个命令的输出当作另一个命令的输入，这种操作需要借助于管道符完成。关于管道的使用在第5章已经进行了讲述，此处不再赘述。

输出重定向使用的符号是“>”，使用输出重定向符的一般形式如下：

```
命令 >文件名
```

该符号类似于比较运算中的大于号，输出重定向符的左边是需要执行的命令，而符号的右边是将左边命令的输出重定向到左边的文件中。如果左边的文件不存在，那么就会创建一个文件，如果文件已经存在，

那么这个文件将被重写，文件中原来的内容将被删除，新的内容是执行的命令的输出内容。如果输出重定向符右边的文件已经存在，那么再次重定向到该文件时，文件原来的内容就会被当前的内容覆盖。

重定向符的一个重要作用就是实现文件的复制，即使用cat命令显示某个文件的内容（源文件），而将cat命令的输出重定向到另外一个文件（目标文件）中，在执行完命令后，就会生成一个复制文件，如示例脚本6-8. sh所示。

```
#示例脚本6-8. sh 输出重定向符

#!/bin/bash

echo 显示数据文件data.txt中的内容
cat data.txt
echo

echo 使用输出重定向符并且显示重定向后的文件内容
cat data.txt >data_bak.txt
cat data_bak.txt
```

为示例脚本6-8. sh赋予可执行权限后执行脚本，结果如下：

```
root@ben-laptop:~/# chmod u+x 6-8. sh
root@ben-laptop:~/# ./6-8. sh
显示数据文件data.txt中的内容
1
```

```
h  
1  
使用输出重定向符并且显示重定向后的文件内容  
1  
h  
1
```

通过脚本6-8. sh的执行结果可以看出，使用输出重定向符能够实现文件的复制。而当重定向的文件不存在时，将创建该文件。而如果文件存在，当前命令的执行结果会将文件中的原内容进行覆盖，从而使得文件只能存储当前命令的执行结果。

为了解决在同一个文件中只能存放当前命令的输出结果的问题，在Shell中提供了对于重定向输出的追加方式，该方式将当前命令的输出内容放到指定文件的后面，这样就避免了当前命令的输出内容替换文件中原来的内容，从而使原文件中的内容不被破坏。

追加重定向使用的操作符是“>>”，其一般形式如下：

```
命令 >>文件名
```

该符号是两个输出重定向符一起使用，在重定向符的中间没有任何符号。而文件可以是存在的文件，也可以是不存在的文件。如果文件名对应的文件存在，那么就以追加的方式将当前命令的执行结果放入文件中；如果文件不存在，那么就将当前命令的执行结果直接放入文件中。使用追加重定向符如示例脚本6-9. sh所示。

```
#示例脚本6-9. sh 追加输出重定向符
#!/bin/bash

echo 显示数据文件data.txt中的内容
cat data.txt
echo

echo 使用输出重定向符并且显示重定向后的文件内容
cat data.txt >data_bak.txt
cat data_bak.txt

echo 使用追加方式进行重定向符并且显示重定向后的文件内容
cat data.txt >>data_bak.txt
cat data_bak.txt
```

为示例脚本6-9. sh赋予可执行权限后执行脚本，结果如下：

```
root@ben-laptop:~/# chmod u+x 6-9.sh
root@ben-laptop:~/# ./6-9.sh
显示数据文件data.txt中的内容
1
h
1
```

使用输出重定向符并且显示重定向后的文件内容

```
1
```

```
h
```

```
1
```

使用追加方式进行重定向符并且显示重定向后的文件内容

```
1
```

```
h
```

```
1
```

```
1
```

```
h
```

```
1
```

通过示例脚本6-9. sh的执行结果可以看出，使用追加重定向符可以保存原文件中的内容，而当前命令的执行结果将会以追加的方式存入已经存在的文件中。

6.4 合并标准输出和标准错误

在使用重定向时，一般一次只会重定向一种文件描述符，但是对于标准输出和标准错误来说，都是输出到屏幕上。因此在重定向时，可以选择将这两个文件描述符一起重定向到某个文件中。一般使用符号“&”来将标准错误和标准输出集合在一起，当重定向以后，标准错误和标准输出同时被重定向，如示例脚本6-10. sh所示。

```
#示例脚本6-10. sh合并标准输出和标准错误
```

```
#! /bin/bash
```

```
echo 合并标准输出和标准错误  
ls /home no_exist  
ls /home no_exist 1>log.txt  
ls /home no_exist 2>log.txt  
ls /home no_exist &>log.txt
```

```
echo 显示重定向文件的内容  
cat log.txt
```

为示例脚本6-10. sh赋予可执行权限后，执行脚本结果如下：

```
root@ben-laptop:~# chmod u+x 6-10. sh  
root@ben-laptop:~# ./6-10. sh  
合并标准输出和标准错误  
ls: 无法访问no_exist: 没有那个文件或目录  
/home:  
ben media  
ls: 无法访问no_exist: 没有那个文件或目录  
/home:  
ben media  
显示重定向文件的内容  
ls: 无法访问no_exist: 没有那个文件或目录  
/home:  
ben  
media
```

通过示例脚本6-10. sh的执行结果可以看出，当指定将输入或输出重定向以后，对应的内容会输出到相应的文件中，而使用符号“&”进行重定向以后，标准输出和标准错误同时被重定向到指定的文件中，而在屏幕上没有任何的输出信息。

6.5 小结

本章主要介绍了Linux系统中文件以及重定向的相关内容。在Linux系统中，所有的内容都被看做是文件，而文件又分为普通文件、设备文件、链接文件等多种类型。使用file或ls命令（添加-l选项）可以查看文件的类型。而文件描述符用来表示打开的所有文件。

在Linux系统中，默认打开的文件为标准输入文件（文件描述符为0）、标准输出文件（文件描述符为1）、标准错误文件（文件描述符为2），标准输入就是从键盘输入数据以供命令、脚本或程序运行使用。标准输出和标准错误则是将输出信息和错误信息输出到显示器上。

为了能够更好、更准确地进行输入和输出，使用重定向将原本标准输入和标准输出进行重定向，输入重定向使用符号“<”表示，而输出重定向使用符号“>”表示。使用重定向以后，原本由键盘输入数据变成了从文件中获取相应的数据，而本来输出到屏幕中的内容也转移到指定的文件中。这样就扩大了命令、脚本、程序的数据来源，保证了数组的正确性和一致性，同时提高了程序的运行效率。

第7章 Shell脚本中的分支结构

在日常生活中经常会遇到选择去做某些事情，而不去做某些事情的情形。如在假期时可以选择去旅游或在家休息，如果去旅游的话，还可以选择去什么地方旅游。在编写Shell脚本时，也存在类似的“选择”结构。当满足某种条件时，可以选择执行某些语句，而在满足另外的条件时，选择执行其他命令。这种选择就是Shell脚本中的分支结构，本章重点介绍分支结构在Shell脚本中的使用。

本章的重点内容如下：

- 条件测试命令的使用
- if分支结构的使用
- case分支结构的使用

7.1 测试命令的使用

在编程语言中，一般使用逻辑表达式作为分支结构的执行条件。当逻辑表达式的结果为逻辑真时，某些语句被执行。当逻辑表达式的结果为逻辑假时，另外的语句被执行。而在Linux中存在一组测试命令，该组命令用于测试某种条件或某几种条件是否真实存在。测试命令是判断语句和循环语句中的条件测试工具，所以，其对于编写Shell非常重要。本小节中将首先介绍测试命令的基础结构，然后介绍测试命令的具体使用。

7.1.1 测试命令的基础结构

在其他的编程语言中，逻辑表达式的结果为真时，返回数值为非0整数（一般为1），而当逻辑表达式的结果为假时，一般返回值为0。而在Shell中，测试命令用来实现测试表达式的条件的真假。如果测试的条件为真，则返回一个0值；如果测试条件为假，将返回一个非0整数值。

在Shell脚本编程中，常用的测试命令有两种形式，一种是用test命令进行测试，其基本形式如下：

```
test expression
```

在上面的结构形式中，条件expression是一个表达式，该表达式可为数字、字符串、文本和文件属性的比较，同时还可以加入各种算术、字符串、文本等运算符，从而实现更加复杂的运算。而test命令用来返回表达式的计算结果。如果计算结果为真，那么返回一个0值，而如果计算结果为假，那么返回值为非0整数值。

注意

test命令的返回结果只能是整数值，而不能是其他的类型。

test命令的使用如示例脚本7-1. sh所示。

```
#示例脚本7-1. sh test命令的使用
```

```
#!/bin/bash

echo 使用test命令进行算术运算
test 3 -le 5
echo "3 -le 5 :$?"

test 5 -le 3
echo "5 -le 3:$?"
```

为示例脚本7-1. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 7-1.sh
ben@ben-laptop:~ $./7-1.sh
使用test命令进行算术运算
3 -le 5 :0
5 -le 3:1
```

通过示例脚本7-1. sh的执行结果可以看出，test命令可以实现算术比较等逻辑运算，当逻辑运算的结果为真时，返回的逻辑结果为0，而如果逻辑运算的结果为逻辑假，返回的结果为1。

注意

在示例脚本7-1. sh中出现的符号-le表示大于等于，该符号和其他类似的符号将在7.1.4小节中进行介绍。

除了使用**test**命令来作为测试命令以外，还有一种结构也可以用来作为测试命令，这种方法的一般形式如下：

```
[ expression ]
```

在上面的结构中，符号“[”是启动测试命令，要求在**expression**后要有一个“]”与其配对。在符号“[”和“]”中间的表达式**expression**和使用**test**命令时的使用方式是一样的。如果表达式**expression**的逻辑结果为真，那么该测试命令的返回值为0，如果该表达式的逻辑运算结果为逻辑假，那么该测试命令的返回值为非0正整数。

命令“[]”的使用如示例脚本7-2. sh所示。

```
#示例脚本7-2. sh命令“[]”的使用
#!/bin/bash

echo 使用[] 代替test命令进行算术运算
[ 3 -le 5 ]
echo "[ 3 -le 5 ] :$?"

[ 5 -le 3 ]
echo "[ 5 -le 3 ]:$?"
```

为示例脚本7-2. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 7-2. sh
ben@ben-laptop:~ $./7-2. sh
```

使用[]代替test命令进行算术运算

```
[ 3 -le 5 ] :0  
[ 5 -le 3 ]:1
```

通过示例脚本7-2.sh的运行结果可以看出，“[]”可以替代test命令完成各种逻辑运算，并且在书写上存在一定的简化。而为了表示test命令和“[]”的不同，在本章中的实例中，将使用test命令和“[]”分别进行脚本的编写和执行。

注意

在使用命令“[]”时，要特别注意“[”后和“]”前的空格必不可少，并且符号“[”后和“]”前都有空格。

7.1.2 测试文件类型

在Linux系统中，所有的内容都被看做文件来处理，因此文件的处理就显得非常重要。对于test命令来说，可以判断文件的某些属性、权限并在文件之间进行各种比较。

使用test命令可以检验文件的相关属性，如被检验的文件是否存在、是否为目录文件、是否是某种类型的文件等，具体的指令以及使用方式如表7-1所示。

表7-1 使用test命令判断文件属性

选项	作用	使用实例
-e	判断文件是否存在	test -e test.txt
-f	判断文件是否存在并且是否是文件 (file)	test -f test.txt
-d	判断文件是否存在并且是否是目录文件 (directory)	test -d test
-c	判断文件是否存在并且是否是字符设备文件	test -c ctest
-S	判断文件是否存在并且是否是Socket文件	test -S Stest
-p	判断文件是否存在并且是否是管道文件 (fifo pipe)	test -p ptest
-L	判断文件是否存在并且是否是链接文件	test -L ltest

从表7-1中可以看出，`test`命令可以判断文件是否存在以及文件属于什么类型。如果文件不存在或者文件的类型与使用的选项不相符，那么`test`命令就会返回逻辑假。如果文件的类型与使用的选项相符合，那么`test`命令就会返回逻辑真。使用`test`命令判断文件属性的实例如示例脚本7-3. sh所示。

```
#示例脚本7-3. sh使用test命令判断文件属性
#!/bin/bash

file1=./7-3. sh
file2=/home
```

```
echo "使用-e选项判断文件是否存在"
test -e file1
echo "test -e $file1 :$?"
test -e file_noexist
echo "test -e file_noexist :$?"

echo "使用-f选项判断文件是否是普通文件"
test -e $file1
echo "test -e $file1:$?"
test -e $file2
echo "test -e $file2 :$?"

echo "使用-d选项判断文件是否是目录文件"
test -d $file1
echo "test -d $file1 :$?"
test -d $file2
echo "test -d $file2:$?"
```

为示例脚本7-3. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 7-3. sh
ben@ben-laptop:~ $./7-3. sh
使用-e选项判断文件是否存在
test -e /7-3. sh:1
test -e file_noexist :1
使用-f选项判断文件是否是普通文件
```

```
test -e ./7-3.sh:0
test -e /home :0
使用-d选项判断文件是否是目录文件
test -d ./7-3.sh :1
test -d /home:0
```

通过实例脚本7-3.sh的执行结果可以看出，在判断文件的属性等基本信息时，可以使用test命令的众多选项来实现相应的判断。

使用test命令除了可以用于判断文件的类型以外，还可以用来判断文件当前的属性，使用不同的选项就可以判断当前文件是否具有该属性，这些选项的用法如表7-2所示。

表7-2 使用test命令判断文件权限

选项	作用	使用实例
-r	判断文件是否具有可读权限	test -r test.txt
-w	判断文件是否具有可写权限	test -w test.txt
-x	判断文件是否具有可执行权限	test -x test
-u	判断文件是否具有SUID权限	test -u ctest
-g	判断文件是否具有SGID权限	test -g Stest
-k	判断文件是否具有 Sticky bit 权限	test -k ptest
-s	判断文件是否存在并且是否是非空白文件	test -s ltest

从表7-2中可以看出，使用test命令可以判断文件是否具有可读、可写、可执行以及SUID、SGID、Sticky bit权限，还能判断文件是否是非空白文件。在使用test命令时，如果文件具有的权限和使用的选项一致，那么test命令执行后就会返回逻辑真（整数0），如果文件具有的权限和使用的选项不一致，那么test命令在执行以后就会返回逻辑假（非0整数）。关于test命令判断文件属性的实例如示例脚本7-4. sh所示。

```
#示例脚本7-4. sh使用test命令判断文件属性
#!/bin/bash

file1=./7-4. sh

ls -l $file1
echo 判断文件是否具有可读权限
test -r $file1
echo "test -r file1 :$?"

echo 判断文件是否具有可写权限
test -w $file1
echo "test -w file1 :$?"

echo 判断文件是否具有可执行权限
test -x $file1
echo "test -x file1 :$?"
```

为示例脚本7-4. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 7-4. sh
ben@ben-laptop:~ $./7-4. sh
-rwxrwxrwx 1 ben ben 294 2014-04-07 11:04 ./7-4. sh
判断文件是否具有可读权限
test -r file1 :0
判断文件是否具有可写权限
test -w file1 :0
判断文件是否具有可执行权限
test -x file1 :0
```

通过示例脚本7-4. sh的执行结果可以看出，使用test命令能够实现判断文件具有什么权限。对于实际应用来说，可以根据实际的需要，对文件不具备的权限进行添加，从而方便用户的使用。

test命令还可以用于对两个文件进行比较，比较的内容则仅是两个文件的新旧以及两个文件是不是同一类型的文件，这些选项的用法如表7-3所示。

表7-3 使用test命令对文件之间的操作

选项	作用	使用实例
-nt	判断 file1 是否比 file2 新	test file1 -nt file2
-ot	判断 file1 是否比 file2 旧	test file1 -ot file2
-ef	判断 file1和file2是否相同	test file1 -ef file2

使用test命令的特定选项就可以判断两个文件的新旧，以及两个文

件是否相同。文件的新旧是按照文件的更新时间和创建时间来确定的。先创建的文件要旧于后创建的，而两个文件只有在类型相同、属性相同并且文件内容相同时，才会相同。关于**test**命令对两个文件的操作实例如示例脚本7-5. sh所示。

```
#示例脚本7-5. sh使用test命令判断文件的新旧
```

```
#!/bin/bash
```

```
file1=/dev
```

```
file2=/dev;
```

```
file3=/tmp;
```

```
echo 使用-nt选项判断文件的新旧
```

```
test $file1 -nt $file3
```

```
echo "$file1 -nt $file3 :$?"
```

```
test $file3 -nt $file1
```

```
echo "$file1 -nt $file3 :$?"
```

```
echo
```

```
echo 使用-ot选项判断文件的新旧
```

```
test $file1 -ot $file3
```

```
echo "$file1 -ot $file3 :$?"
```

```
test $file3 -ot $file1
```

```
echo "$file1 -ot $file3 :$?"
```

```
echo
```

```
echo 使用-ef选项判断文件是否相同
```

```
test $file1 -ef $file2
echo "$file1 -ot $file3 :$?"
test $file2 -ef $file3
echo "$file1 -ot $file3 :$?"
```

为脚本7-5. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 7-5. sh
```

```
ben@ben-laptop:~ $./ 7-5. sh
```

使用-nt选项判断文件的新旧

```
/dev -nt /tmp :1
```

```
/dev -nt /tmp :0
```

使用-ot选项判断文件的新旧

```
/dev -ot /tmp :0
```

```
/dev -ot /tmp :1
```

使用-ef选项判断文件是否相同

```
/dev -ot /tmp :0
```

```
/dev -ot /tmp :1
```

通过实例脚本7-5. sh的执行结果可以看出，使用test命令可以实现对文件的新旧进行比较，并且能够判断文件是否相同，在实际应用中，可以在对文件操作时，选择新的版本还是旧的版本，从而达到对数据的精确要求。

7.1.3 测试字符串

在编写Shell脚本时，经常操作的对象还包括字符串，对字符串的操作包括测试字符串是否为空、两个字符串是否相等，常用于测试用户输入的是否为空或比较字符串变量是否是某个特定值。常用的字符串运算符如表7-4所示。

表7-4 常用字符串测试运算符

符号	作用	使用实例
-n string	测试字符串string是否不为空	test -n "hello"
-z string	测试字符串string是否为空	test -z "hello"
string1=string2	测试字符串string1是否与字符串string2相同	test "hello"="hello"
string1!=string2	测试字符串string1是否与字符串string2不相同	test "hello"! ="hello"

通过表7-4中可以看出，使用test命令对字符串的操作只有判断测试的字符串是否为空以及字符串是否是某个值。对于-n选项来说，都是判断字符串是否为空，如果字符串为空，那么test命令就会返回逻辑假的值。如果字符串不为空，就返回逻辑真的值。而对于选项-z来说，返回的结果和-n选项正好相反。对于相同符号“=”和不相同符号“!=”来说，只要是符号左右两边的字符串相同（或不同），那么就返回逻辑真（假）的结果。关于字符串测试运算符的使用实例如示例脚本7-6. sh所示。

```
#示例脚本7-6. sh使用test命令处理字符串
```

```
#! /bin/bash
```

```
str1="hello";
```

```
str2="Hello";
```

```
str_null=""
```

```
str3="hello";
```

```
echo 使用-n选项测试字符串是否为空
```

```
echo 测试字符串str1是否为空
```

```
test -n $str1
```

```
echo "test -n $str1:$?"
```

```
echo 测试字符串str2是否为空
```

```
test -n $str2
```

```
echo "test -n $str2:$?"
```

```
echo 使用-z选项测试字符串是否为空
```

```
test -z $str_null;
```

```
echo "test -z $str_null:$?"
```

```
echo 使用符号=判断两个字符串是否相同
```

```
test $str1 = $str2;
```

```
echo "$str1 = $str2 :$?"
```

```
test $str1 = $str3;
```

```
echo "$str1 = $str3 :$?"
```

```
echo 使用符号!=判断两个字符串是否相同
```

```
test $str1 != $str2;
echo "$str1 = $str2:$?"
test $str1 != $str3;
echo "$str1 = $str3 :$?"
```

为脚本7-6. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 7-6. sh
ben@ben-laptop:~ $./ 7-6. sh
使用-n选项测试字符串是否为空
测试字符串$str1是否为空
test -n hello:0
测试字符串$str2是否为空
test -n Hello:0
使用-z选项测试字符串是否为空
test -z :0
使用符号=判断两个字符串是否相同
hello = Hello :1
hello = hello :0
使用符号!=判断两个字符串是否相同
hello = Hello:0
hello = hello :1
```

通过脚本7-6. sh的执行结果可以看出，使用字符串运算符号，可以判断字符串是否为空以及两个字符串是否相同。而使用不同的选项对

字符串进行测试，就会得到不同的结果。

注意

比较字符串时建议字符串变量使用双引号，即使变量为空，也要使用双引号。

7.1.4 测试数值

对于数值的测试主要是对整数的比较，主要包括整数之间的大小判断、是否相等的判断，常用的整数比较运算符如表7-5所示。

表7-5 常用整数比较运算符

符号	作用	使用实例
num1 -eq num2	num1和num2是否相等	test 3 -eq 5
num1 -ge num2	num1是否大于或等于num2	test 3 -ge 5
num1 -gt num2	num1是否大于num2	test 3 -gt 5
num1 -le num2	num1是否小于或等于num2	test 3 -le 5
num1 -lt		

num2	num1是否小于num2	test 3 -lt 5
num1 -ne num2	num1是否不等于num2	test 3 -ne 5

通过表7-5可以看出，使用特定的符号可以实现整数的比较，可以实现数值的大小比较，这些符号的使用方式如示例脚本7-7. sh所示。

```
#示例脚本7-7. sh使用test命令判断数值是否相等
#!/bin/bash

num1=3
num2=5
num3=5

echo 判断两个数是否相等
test $num1 -eq $num2
echo "$num1 = $num2 :$?"
echo

echo 判断两个数的大小
test $num3 -le $num2
echo "$num3 = $num2 :$?"
echo

test $num3 -lt $num2
echo "$num3 -lt $num2 :$?"
```

为脚本7-7. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 7-7.sh  
ben@ben-laptop:~$./ 7-7.sh  
判断两个数是否相等  
3 = 5 :1  
  
判断两个数的大小  
5 = 5 :0  
  
5 -lt 5 :1
```

通过示例脚本7-7. sh的执行结果可以看出，在进行数值的比较时，使用不同的符号进行判断，得到的结果就不会相同。

注意

不能使用大于号或小于号，否则bash会误认为是重定向符号。

7.1.5 复合测试条件

前面的小节中所使用的逻辑表达式都是单个逻辑表达式，而在实际的应用中，经常出现的情况是同时满足多个条件或满足多个条件中的某几个条件，才会执行相应的操作。这样就需要使用逻辑运算符将几个逻辑表达式进行连接。常用的逻辑运算符如表7-6所示。

表7-6 常用逻辑运算符

运算符名称	作用	使用实例
!	如果expression为假，则测试结果为真	! expression
-a	如果expression1和expression2同时为真，则测试结果为真	expression1-a expression2
-o	如果expression1和expression2有一个为真，则测试结果为真	expression1-o expression2

从表7-6中可以看出，Shell脚本中的逻辑运算符和其他编程语言中使用的逻辑运算符的使用方式是相同的，但是表示方式有所不同。在Shell脚本中，使用于感叹号“！”来表示逻辑非。当表达式expression为真时，返回的结果为逻辑假，而当表达式expression的运算结果为假时，得到的最终结果才为真。对于-a选项来说，只有两个表达式同时为真，最终的结果才是真，如果其中的一个表达式的结果为假，那么test命令就会返回逻辑假。而对于-o选项来说，只有当所有的表达式都为假时，test命令才会返回逻辑假，除此之外，都会返回逻辑真。关于逻辑运算符的使用方式如示例脚本7-8. sh所示。

```
#示例脚本7-8. sh复合测试条件的使用
```

```
#!/bin/bash
```

```
num1=3
```

```
num2=5
```

```
num3=5
```

```
echo 使用逻辑或运算符  
test $num1 -ge $num2 -o $num3 -eq $num2  
echo "逻辑或运算结果:$?"  
echo
```

```
echo 使用逻辑与运算符  
test $num1 -ge $num2 -a $num3 -eq $num2  
echo "逻辑与运算结果:$?"
```

为脚本7-8. sh赋予可执行权限后运行脚本，操作步骤如下：

```
ben@ben-laptop:～ $chmod u+x 7-8. sh  
ben@ben-laptop:～$./ 7-8. sh  
使用逻辑或运算符  
逻辑或运算结果:0  
  
使用逻辑与运算符  
逻辑与运算结果:1
```

通过示例脚本7-8. sh的执行结果可以看出，在需要使用多个逻辑运算表达式时，可以使用逻辑运算符将多个表达式连接在一起，按照逻辑运算符的运算规则进行运算，从而得出最终的逻辑运算结果。

注意

逻辑非运算符和表达式之间要有空格。

7.2 if分支结构

在Shell脚本中，也存在用于处理分支结构的语句，最基本的结构类型就是if结构。当使用if结构时，只有满足某个条件才会执行某些语句，而在不满足此条件的情况下则会执行其他的语句。本节将介绍if分支结构的使用方式。

7.2.1 if-then结构

在分支结构中，最基本的是if -then结构。if -then语句的基本格式如下：

```
if expression  
then  
commands  
fi
```

在上述结构中，expression是一个逻辑表达式，可以使用test命令表示，也可以使用测试命令“[]”表示。expression可以是一个逻辑表达式，也可以是多个逻辑表达式的有机组合。当表达式的逻辑运算结果为真时，才会执行then与fi中间的commands。如果得到的结果不是逻辑真，那么commands就不会被执行，而去执行fi后面的语句。在使用if-then结构时，不要忘记使用fi作为结构的结束符，否则在脚本执行时，会提示

有错误。关于if-then结构的使用实例如示例脚本7-9. sh所示。

```
#示例脚本7-9. sh  if结构使用实例
#!/bin/bash

echo 使用test命令进行逻辑判断
echo 判断3是否小于5
if test 3 -le 5
then
    echo 3 小于 5
fi
echo
echo 判断5是否小于3
if test 5 -le 3
then
    echo 5 小于 3
fi
```

为示例脚本7-9. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x7-9. sh
ben@ben-laptop:~$./ 7-9. sh
使用test命令进行逻辑判断
判断3是否小于5
3 小于 5
```

判断5是否小于3

示例脚本7-9. sh展示了使用test命令进行3与5数字大小的逻辑判断，输出相应的提示性信息。下面将使用测试命令“[]”来重新实现示例脚本7-10. sh，具体的内容如下所示。

```
#示例脚本7-10. sh 使用符号“[]”代替test命令
#!/bin/bash

echo 使用[] 命令替换test进行逻辑判断
echo 判断3是否小于5
if [ 3 -le 5 ]
then
    echo 3 小于 5
fi

echo 判断5是否小于3
if [ 5 -le 3 ]
then
    echo 5 小于 3
fi
```

为示例脚本7-10. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x7-10. sh
ben@ben-laptop:~$ ./ 7-10. sh
```

```
使用[] 命令替换test进行逻辑判断
```

```
判断3是否小于5
```

```
3 小于 5
```

```
判断5是否小于3
```

通过示例脚本7-10. sh的执行结果可以看出，使用命令“[]”可以代替test命令来作为逻辑表达式，从而作用于if-then结构。

7.2.2 if-then-else结构

示例脚本7-9. sh和7-10. sh中的if-then结构仅能在表达式成立的时候才会执行相应的语句，而当表达式不成立时，没有执行特定的表达式，而是执行了结构外面的语句。为了使得编写的脚本逻辑性更加清晰，可以使用if -then-else结构来代替if -then结构，这样即使表达式不成立，也会有特定的语句被执行，从而使得结构的逻辑性更加清晰。if -then-else结构的基本格式如下：

```
if expression  
then  
    commands1  
else  
    commands2  
fi
```

在上述结构中，expression是一个逻辑表达式，可以使用test命令表

示，也可以使用测试命令“[]”表示。expression可以是一个逻辑表达式，也可以是多个逻辑表达式的有机组合。当表达式的逻辑运算结果为真时，才会执行语句commands1。如果得到的结果是逻辑假，那么commands2就会被执行。当commands1或commands2执行完毕后，才会执行fi后面的语句。由此可以看出，使用if-then-else结构增加了当表达式expression结果为逻辑假时的处理，这样显得脚本的逻辑性更加清晰。关于if-then-else结构的使用实例如示例脚本7-11. sh所示。

```
#示例脚本7-11. sh if-then-else结构
#!/bin/bash

echo 使用test命令进行逻辑判断
echo 判断3是否小于5
if test 3 -le 5
then
    echo 3 小于 5
else
    echo 3 大于 5
fi

echo 判断5是否小于3
if test 5 -le 3
then
    echo 5 小于 3
else
    echo 5 大于 3
fi
```

为示例脚本7-11. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x7-11. sh
ben@ben-laptop:~$ ./ 7-11. sh
使用test命令进行逻辑判断
判断3是否小于5
3 小于 5
判断5是否小于3
5 大于 3
```

if-then-else结构也可以使用测试命令“[]”来表示expression，从而得出最终的逻辑结果。使用测试命令“[]”表示if-then-else结构的实例如示例脚本7-12. sh所示。

```
#示例脚本7-12. sh 命令“[]”在if-then-else结构中的使用
#!/bin/bash

echo 使用[] 命令替换test进行逻辑判断
echo 判断3是否小于5
if [ 3 -lt e 5 ]
then
    echo 3 小于 5
else
    echo 3 大于 5
fi
```

```
echo 判断5是否小于3
if [ 5 -le 3 ]
then
    echo 5 小于 3
else
    echo 5 大于 3
fi
```

为示例脚本7-12. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x7-12. sh
ben@ben-laptop:~$./ 7-12. sh
使用[] 命令替换test进行逻辑判断
判断3是否小于5
3 小于 5
判断5是否小于3
5 大于 3
```

通过示例脚本7-11. sh 和7-12. sh可以看出，if-then-else结构比if-then结构在逻辑上更加清晰和完善，当表达式成立时可以执行某些语句，而表达式不成立时，也可以执行相关的语句，从而保证逻辑的完整性。

7.2.3 嵌套结构

在其他的编程语言中，都存在嵌套结构。而所谓的嵌套结构就是指在某个结构中还存在该结构，从而形成该结构的嵌套使用。对于Shell脚本中的分支结构来说，if-then和if-then-else结构都存在嵌套结构。if-then结构的嵌套基本形式如下：

```
if expression1  
then  
if expression2  
then  
commands1  
fi  
commands2  
fi
```

在该结构中，首先判断逻辑表达式expression1的逻辑结果是逻辑真还是逻辑假。如果得到的结果为逻辑真，那么按照if-then结构的执行方式来执行内层的if-then结构。执行完内层的if-then结构以后，再执行外层if-then结构中的commands2。当commands2执行完毕后，整个if-then结构才算执行完毕。关于if-then结构的嵌套使用实例如脚本7-13. sh所示。

```
#示例脚本7-13. sh 命令“[]”在if-then-else结构中的使用  
#!/bin/bash  
  
echo if-then结构的嵌套使用  
echo 判断3和5的关系
```

```
if [ 3 -le 5 ]
then
    if [ 3 -eq 5 ]
    then
        echo 3 等于 5
    fi

    if [ 3 -ge 5 ]
    then
        echo 3 大于 5
    fi
echo 3 小于 5
fi
```

为示例脚本7-13. sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x7-13. sh
ben@ben-laptop:~$ ./ 7-13. sh
if-then结构的嵌套使用
判断3和5的关系
3 小于 5
```

通过实例脚本7-13. sh的执行结果可以看出，使用嵌套结构能够使得程序的逻辑判断更加详细和完善，但是如果判断条件较多时会发生混乱的情形，为了解决这种问题，还可以使用if-elif-else结构来代替这种嵌套结构，该结构的基本形式如下：

```
if expression1
then
    command1
elif expression2
then
    commands2
else
    command3
fi
```

在上面的结构中，首先判断表达式expression1的逻辑运算结果。如果表达式expression1的运算结果为逻辑真，那么将执行语句command1。当执行完毕后，整个结构也执行完毕。而如果express1的运算结果为逻辑假，那么被执行的语句将判断表达式elif后面的表达式expression1是否成立。如果表达式expression2成立，那么就执行command2。如果所有的表达式都不成立，将会执行else部分中的command3。在if-elif-else结构中可以出现多个elif结构，其执行顺序也是从上向下依次执行。其执行结构如图7-1所示。

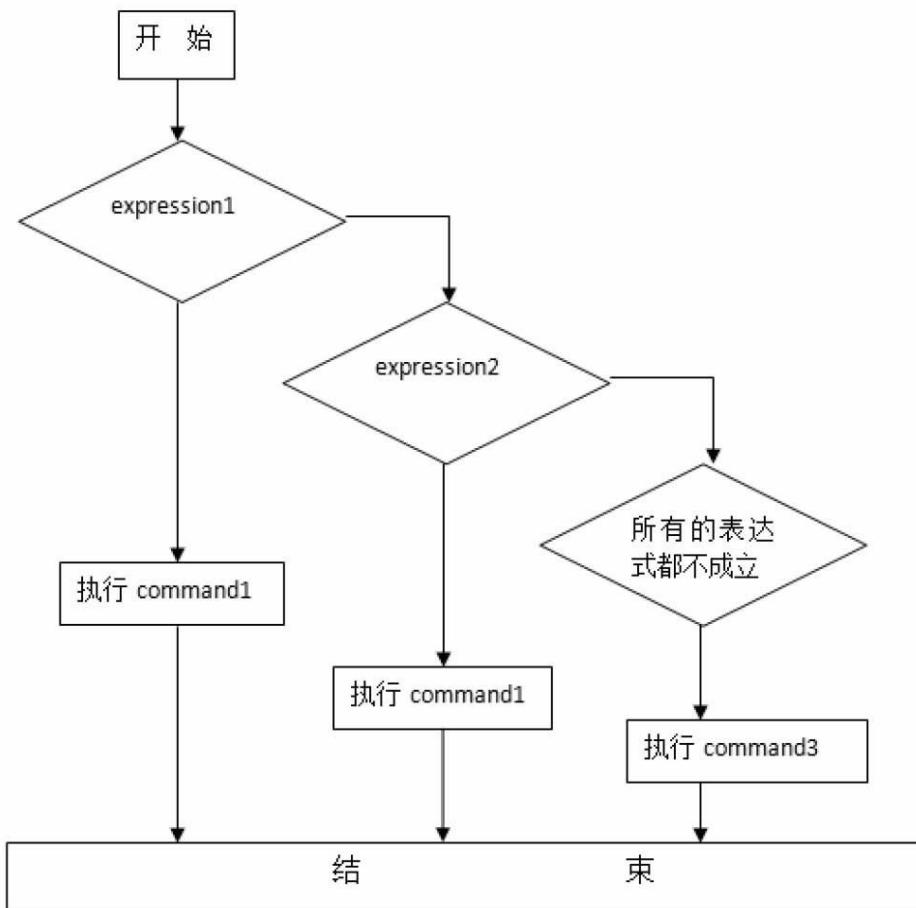


图7-1 if-elif-else执行顺序示意图

使用if-elif-else结构重新编写示例脚本7-13. sh，其内容如示例脚本7-14. sh所示。

```

#示例脚本7-14. sh  if-elif-else结构的使用
#!/bin/bash

echo if-then结构的嵌套使用
echo 判断3和5的关系
if [ 3 -le 5 ]
then

```

```
echo 3小于5
elif [ 3 -eq 5 ]
then
echo 3等于5
else
echo 3大于5
fi
```

通过示例脚本7-14. sh可以看出，if-elif-else结构的使用比嵌套使用if-then-else结构更加灵活，用户可以选择嵌套使用if-then结构，也可以嵌套调用if-then-else结构，还可以使用if-elif-else结构。而在任意的结构中，还可以嵌套使用其他的结构。此时，可供用户选择的方式就变得很多，这就需要在具体的使用时根据实际需要选择合适的嵌套方式。

7.3 case多条件分支结构

在Shell脚本中处理分支结构时除了使用if结构以外，还可以使用case结构。case结构一般用于在一组值当中寻找特定的值的情形，比if结构能够表示的范围要窄，但是在某些情形下，使用case结构比使用if结构更适合。本小节将重点讲述如何在编写Shell脚本时使用case结构。

7.3.1 case结构基础

case结构一般用来匹配特定的数值，并且需要按照一定的格式进行编写，其基本格式如下：

```

case 变量 in
  pattern1)  commands1;;
  pattern2| pattern3) command2;;
*)  command3;;
esac

```

在上面的结构中，变量的数值匹配case结构中的pattern，如果匹配成功，那么就执行对应的命令，如果匹配不成功，那么就匹配其他的pattern。当所有的都不匹配时，将执行星号“*”中的命令。该结构的执行顺序如图7-2所示。

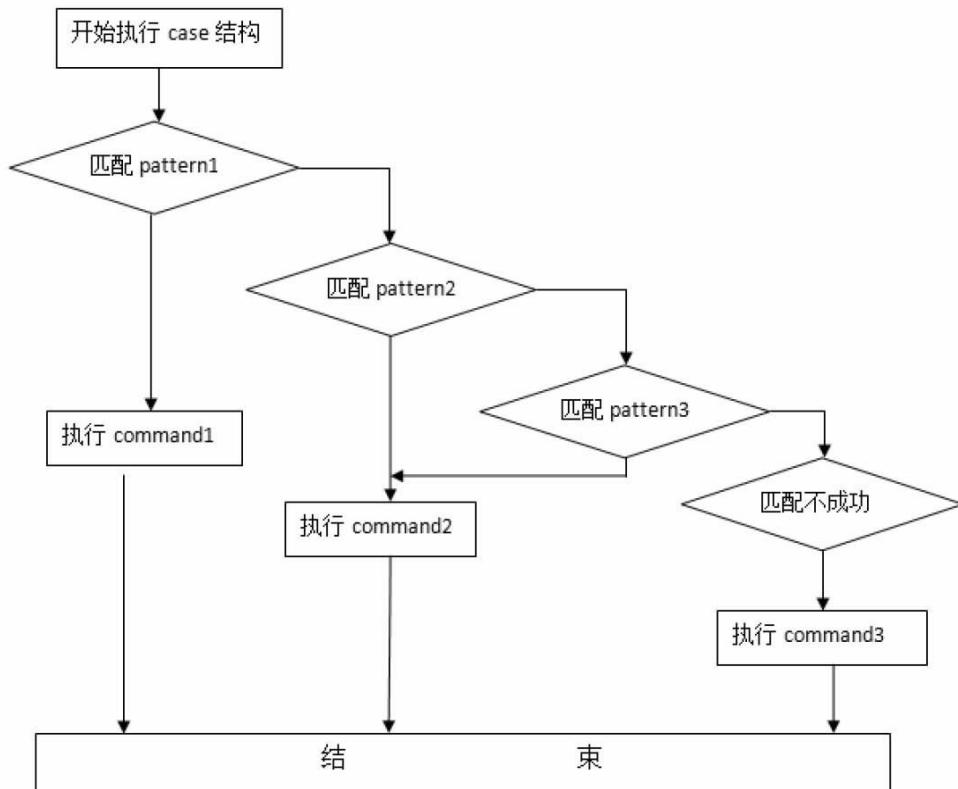


图7-2 case结构执行顺序示意图

从图7-2中可以看出，执行case结构时，首先匹配pattern。如果匹配

成功，则会执行对应的command。还可以使用或运算符“|”来表示匹配pattern2成功或匹配pattern3成功，都可以执行command2。而当所有的匹配数值都不能成功时，还需要执行默认的命令，从而使case结构变得完整。使用case结构的实例如示例脚本7-15所示。

```
#示例脚本7-15. sh case结构的使用
#!/bin/bash

echo -n 输入一个分数：
read num

case $num in
    1) echo 输入的数值为1;;
    2) echo 输入的数值为2;;
    3) echo 输入的数值为3;;
    4) echo 输入的数值为4;;
    5) echo 输入的数值为5;;
    *) echo 输入的数值大于5;;
esac

echo case 结构运行结束
```

为示例脚本7-15.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x7-15.sh
ben@ben-laptop:~$ ./ 7-15.sh
输入一个分数:3
```

```
输入的数值为3
case 结构运行结束
ben@ben-laptop:~ $ ./7-15*
输入一个分数:1
输入的数值为1
case 结构运行结束
ben@ben-laptop:~ $ ./7-15*
输入一个分数:7
输入的数值大于5
case 结构运行结束
```

通过示例脚本7-15.sh 的执行结果可以看出，在使用**case**结构时，根据输入的内容的不同，**case**结构会自动执行匹配成功的分支中的指令，而对于其他的指令则“置之不理”，如果都没有匹配成功，就直接执行默认的结构。

注意

在使用**case** 结构时，在结尾处要添加**esac** 作为**case** 结构的结尾，在每一条执行的后面需要添加两个分号作为分支执行结束的标志。

7.3.2 在Shell 脚本中使用**case** 结构

case结构除了使用整型数值之外，在Shell脚本中还可以匹配其他类

型的数值，如字符串等。具体参见示例脚本7-16.sh。

```
#示例脚本7-16.sh case 结构的使用
#!/bin/bash
echo 当前使用的用户为$USERNAME
echo 使用case 结构
case $USERNAME in
    "root") echo 使用root 用户登录，具有最高权限;;
    "ben") echo 使用ben 用户登录，具有普通权限;;
    *) echo 使用其他用户登录，具有权限不明确;;
esac
echo 结束case结构
```

为示例脚本7-16.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x7-16.sh
ben@ben-laptop:~$ ./ 7-16.sh
当前使用的用户为ben
使用case结构
使用ben用户登录，具有普通权限
结束case结构
```

根据示例脚本7-16.sh的执行结果可以看出，在Shell脚本中使用case结构时，匹配的变量可以是整数，也可以是某个变量，还可以是某个命令的执行结果。可以根据实际需要选择case结构来实现分支结构的处理。

`case`结构可以使用`if`结构表示，对于示例脚本7-16.sh来说，可以使用`if-then`结构表示，如示例脚本7-17.sh所示。

```
#示例脚本7-17.sh 使用if结构替换case结构
#!/bin/bash

echo 使用if结构替换case结构
echo 当前使用的用户为$USERNAME

if [ $user = root ]
then
echo 使用root用户登录，具有最高权限

elif [ $user = ben ]
then
echo 使用ben用户登录，具有普通权限
else
echo 使用其他用户登录，具有权限不明确
fi
```

为示例脚本7-17.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x7-17.sh
ben@ben-laptop:~$ ./ 7-17.sh
使用if结构替换case结构
当前使用的用户为ben
```

使用ben用户登录，具有普通权限

通过示例脚本7-17.sh的执行结果可以看出，使用if-then结构可以直接代替case结构来实现分支结构类型的脚本的编写，因此，在使用分支类型的脚本时，可以根据实际需要来选择使用case结构还是if-then结构。

7.3.3 select命令的使用

在使用case结构时，需要提前输入用户的操作方式然后匹配做什么操作。在Shell脚本中，还可以使用select命令来实现自动地获取用户的输入并且自动根据用户的输入进行处理。select命令的一般使用方式如下：

```
select 变量 in 变量列表
do
    命令
done
```

在使用select命令时，将在变量列表中查找变量，如果在列表中找到对应的值，就会执行相应的命令，找不到的话就不会执行该命令后面的语句。select命令将变量列表中的每一项都作为一个编号来显示，用户可以根据选项输入适当的内容，使用select命令的实例如示例脚本7-18.sh所示。

```
#示例脚本7-18.sh select命令的使用实例
#!/bin/bash

echo select命令的使用

select choose in 鸡 鸭 鱼 肉 其他
do
break
done

echo 你喜欢的是$choose
```

为示例脚本7-18.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 7-18.sh
ben@ben-laptop:~$./7-18.sh
select命令的使用
1) 鸡
2) 鸭
3) 鱼
4) 肉
5) 其他
#? 3
你喜欢的是鱼
```

通过示例脚本7-18.sh的执行结果可以看出，使用select命令可以首先显示菜单，然后能自动根据用户的输入获取选项的内容。虽然输入的是数字，但是输出的却是数字表示的字符串。

注意

`select`命令可以循环获取用户的输入，但是在第二次以后就不再显示菜单，因此，一般不使用`select`的循环特性。

7.4 小结

本章主要介绍了测试命令的使用以及常用的分支结构：`if`分支结构和`case`分支结构的使用方式。测试命令包括`test`命令和“[]”命令两种形式。

使用测试命令可以实现对文件类型、字符串、数值的各种判断，如判断文件是普通文件还是目录文件、具有什么样的权限等；对于字符串的判断包括字符串是否相同、是否为空等常用操作；而`test`命令和“[]”命令可以进行互换。值得注意的是，在Shell脚本中，如果测试条件的结果为真，则返回一个0值；如果测试条件的运算结果为假，将返回一个非0整数值。

使用`if`结构可以实现分支结构的操作，满足某个条件才会执行某些语句，而在不满足此条件的情况下则会执行其他的语句。常用的`if`结构包括`if-then`结构、`if-then-else`结构以及`if-elif-then`结构。这3种结构还可以进行互相的嵌套使用，从而满足在实际编写脚本时对分支结构的需要。

除了使用`if`结构实现Shell脚本中的分支结构以外，还可以使用`case`结构实现。`case`结构能够在既定的字符序列中选择匹配项的命令执行，对于不匹配的内容则不去执行。在实际操作过程中，可以按照实际需要

选择使用哪种操作方式。而在某些情况下，可以使用select命令来替换case结构。select命令能够自动获取用户的输入并且自动根据用户的输入进行处理。

第8章 Shell中的循环结构

在日常应用中，除了根据某些条件选择去做什么事情以外，还经常会根据某些条件不断地去做某些事情，如在到了9点以后我们就去上班，在上班的时候我们会不断地去工作，而到了下班的时候则不再工作。这个简单的例子就说明了在满足特定的条件（到了上班的时间点）时，不断地去做某些事情（完成各种工作）。在Shell脚本中，除了条件结构以外，也同样存在循环结构。循环结构是在满足一定条件下重复执行某些操作而使用的结构，而在特定的条件下，循环结构能够主动退出，而不会产生不断执行循环结构的情形。本章将重点介绍在Shell脚本中如何实现循环结构。

本章的主要内容如下：

- `for`循环结构的使用
- `while`循环结构的使用
- 循环嵌套的使用
- 循环控制符`break`、`continue`的使用

8.1 `for`循环

在高级编程语言中，循环结构一般是指`for`结构和`while`结构，而在Shell脚本中，也可以使用`for`结构和`while`结构来实现循环处理。本小节将重点介绍`for`结构的使用方式。

8.1.1 使用for-in结构

在Shell脚本中，for循环结构一般与in关键字共同使用。for循环结构的基本形式如下：

```
for 变量 in 取值列表
do
    commands;
done
```

在上面的结构中，for命令后面的变量需要到取值列表中依次取值，并且变量可以用于循环结构中。取值列表是变量的取值范围，一般是一组用空格分隔的字符串，而每次进行for...in 读取时，就会按顺序将读到的值赋给前面的变量，直到最后一个字符串参与脚本的执行。如果变量的取值在取值列表中，就会执行do命令和done命令中间的commands。执行完一次后，就会到取值列表中取下一个值，直到最后一个值结束。取值列表for-in结构的使用实例如示例脚本8-1.sh所示。

```
#示例脚本8-1.sh for结构的使用
#!/bin/bash

echo "for-in结构的简单示例"
for str in spring summer autumn winter
do
    echo " the season is $str"
done
```

为示例脚本8-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-1.sh
ben@ben-laptop:~$ ./8-1.sh
the season is spring
the season is summer
the season is autumn
the season is winter
```

在for-in结构中，取值列表默认使用空格进行分隔，而如果使用的某个字符串需要多个字符串结合在一起使用，那么就需要使用双引号将需要的字符串引起来，从而作为一个字符串处理，如示例脚本8-2.sh所示。

```
#示例脚本8-2.sh 变量列表的特殊使用
#!/bin/bash

echo "不使用双引号"
for str in fish meat Italian Pizza
do
    echo "今天晚上的菜是： $str"
done

echo "使用双引号"
for str in fish meat "Italian Pizza"
do
```

```
echo "今天晚上的菜是: $str"  
done
```

为示例脚本8-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-2.sh  
ben@ben-laptop:~$ ./8-2.sh  
不使用双引号  
今天晚上的菜是: fish  
今天晚上的菜是: meat  
今天晚上的菜是: Italian  
今天晚上的菜是: Pizza  
使用双引号  
今天晚上的菜是: fish  
今天晚上的菜是: meat  
今天晚上的菜是: Italian Pizza
```

通过示例脚本8-2.sh的执行结果可以看出，如果两个单词是作为一个变量来处理，如果不使用引号将这两个单词引起来，那么就会被当做一个单词。因为在Shell脚本中，变量与变量之间是通过空格来分隔的。

注意

取值列表中出现的双引号不会出现在字符串变量中，其作用仅是说明引号之间的字符串是一个字符串。

在使用for-in结构时，变量的取值不一定非得限制在取值列表中，在脚本中可以进行任意的改变。可以在循环结构中赋予其他的值，在循环结构结束后，也可以进行各种操作，如示例脚本8-3.sh所示。

```
#示例脚本8-3.sh 变量列表的特殊使用
#!/bin/bash
echo "at beginning, the num = $num"
for num in 1 2 3 4 5 6 7
do
    $num=$num+1
    echo "the change, the num = $num"
done
$num=100;
echo "after for in , the num = $num"
```

为示例脚本8-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-3.sh
ben@ben-laptop:~$ ./8-3.sh
实例脚本$ ben@ben-laptop:~$ ./8-3*
at the begining, the num=1
the change, the num = 1
the change, the num = 2
the change, the num = 3
the change, the num = 4
the change, the num = 5
```

```
the change, the num = 6  
the change, the num = 7  
  
after for in , the num = 100
```

在日常操作for-in结构时，利用for-in结构的特性还可以处理某个目录文件中的所有文件，如使用test命令判断其中的文件是什么类型以及文件的权限等。在处理这方面的问题时，只需要将取值列表换成需要处理的文件目录即可，如示例脚本8-4.sh所示。

```
#示例脚本8-4.sh 使用目录作为变量列表  
#!/bin/bash  
  
for file in $(ls /home)  
do  
    if test -d $file  
    then  
        echo $file是目录文件  
    elif test -f $file  
    then  
        echo $file 是普通文件  
    elif test -L $file  
    then  
        echo $file是链接文件  
    elif test -S $file  
    then
```

```
        echo $file 是socket文件
else
        echo $file 未知类型文件
fi
done
```

为示例脚本8-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-4.sh
ben@ben-laptop:~$ ./8-4.sh
ben 是目录文件
media是目录文件
```

通过脚本8-4.sh的执行结果可以看出，在for循环结构中，不但可以使用普通的变量列表，还可以使用命令执行的结果来作为变量的取值，从而扩展了for结构的使用范围。

注意

在示例脚本8-4.sh中进行文件名检测的时候，参数列表还可以使用其他方式，有兴趣的读者可以进行测试。

8.1.2 C式for结构

在C语言编程中也有类似的for结构，但是在C语言中，for结构的基本形式是：

本结构如下：

```
for (expression1; expression2; expression3)
{
    循环体;
}
```

在上述结构中，expression1一般用于变量的初始化，而expression2则是一个逻辑表达式，当该逻辑表达式的运行结果为真时，for循环中的循环体部分才会被执行，而当该表达式的运行结果为逻辑假时，直接跳过for循环，而去执行for循环之外的语句。expression3是一个循环变量变化的表达式，用来对循环变量进行各种变化操作。

在Shell脚本中存在类似的for结构，其基本形式如下：

```
for (( expression1; expression2; expression3 ))
do
    循环体;
done
```

在上面的结构中，expression1也是给变量附赋初始值，赋值语句不像是Shell中的其他赋值语句一样，在等号的两边可以有空格。
expression2是一个逻辑表达式，当逻辑表达式的结构为逻辑真时，循环体部分才会被执行，而当逻辑表的结果为逻辑假时，将直接跳出循环体，而去执行循环体外面的语句。expression3是一个使得变量发生变化的表达式，通过这个表达式可以使得变量发生变化。在表达式expression2和expression3中，对于变量的引用不必使用美元符号“\$”，而

直接使用变量名即可。C式for命令的使用实例如示例脚本8-5.sh所示。

```
#示例脚本8-5.sh C式for命令
#!/bin/bash

echo "使用C式for命令"
for (( i = 1; i <= 10; i++ ))
do
    echo "当前的变量值为: $i";
done
```

为示例脚本8-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-5.sh
ben@ben-laptop:~$ ./8-5.sh
"使用C式for命令"
"当前的变量值为: 1"
"当前的变量值为: 2"
"当前的变量值为: 3"
"当前的变量值为: 4"
"当前的变量值为: 5"
"当前的变量值为: 6"
"当前的变量值为: 7"
"当前的变量值为: 8"
"当前的变量值为: 9"
```

“当前的变量值为： 10”

通过脚本8-5.sh的执行结果可以看出，使用C式for命令能够实现类似于C语言中的for循环结构的功能，从而方便熟悉C语言的用户来编写相应的for结构，简化了Shell脚本中的for循环结构。

C式for命令虽然大部分类似于C语言中的for结构，但是也存在一定的差别。这个差别就在于在Shell中，C式for结构中expression2中的逻辑表达式只能有一个，而不像在C语言中那样，存在多个逻辑表达式。但是可以像在C语言中那样存在多个变量，如示例脚本8-6.sh所示。

```
#示例脚本8-6.sh 使用多个表达式
#!/bin/bash

echo "使用C式for命令包含一个逻辑表达式"
for (( i = 1; i <= 10; i++ ))
do
    echo "当前的变量值为: $i";
done

echo "使用C式for命令包含两个变量以及逻辑表达式"
for (( i = 1, j = 5; i <= 10; i++, j-- ))
do
    echo "变量i的当前值为: $i";
    echo "变量j的当前值为: $j";
done
```

为示例脚本8-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-6.sh
```

```
ben@ben-laptop:~$ ./8-6.sh
```

使用C式for命令包含一个逻辑表达式

当前的变量值为： 1

当前的变量值为： 2

当前的变量值为： 3

当前的变量值为： 4

当前的变量值为： 5

当前的变量值为： 6

当前的变量值为： 7

当前的变量值为： 8

当前的变量值为： 9

当前的变量值为： 10

使用C式for命令包含两个变量以及逻辑表达式

变量i的当前值为： 1

变量j的当前值为： 5

变量i的当前值为： 2

变量j的当前值为： 4

变量i的当前值为： 3

变量j的当前值为： 3

变量i的当前值为： 4

变量j的当前值为： 2

变量i的当前值为： 5

变量j的当前值为： 1

```
变量i的当前值为: 6
变量j的当前值为: 0
变量i的当前值为: 7
变量j的当前值为: -1
变量i的当前值为: 8
变量j的当前值为: -2
变量i的当前值为: 9
变量j的当前值为: -3
变量i的当前值为: 10
变量j的当前值为: -4
```

通过示例脚本8-6.sh的执行结果可以看出，在Shell脚本中，也可以使用多个变量，在进行处理时，每一个变量都会按照执行顺序进行，直至逻辑表达式的运算结果为假，才会退出整个循环。

注意

C式的for命令方便了熟悉C语言的程序员，由于其语法的改变，对于熟悉Shell编程的程序员来说，却是需要格外注意的。

8.2 while命令的使用

在编写Shell脚本中的循环结构时，除了使用for循环结构以外，还可以使用while结构。while结构更像是if-then结构和for结构的组合，即只有满足特定条件时才会执行循环结构。本小节重点介绍while结构以

及类似的until结构的使用方式。

8.2.1 使用while结构

while结构的基本格式如下：

```
while test expression
do
    commands
done
```

通过上面的基本格式可以看出，在执行while结构时，首先使用test命令判断表达式expression的结果。expression是一个逻辑表达式，当表达式的逻辑结果是逻辑真时，while循环结构中的commands才会被执行。而当表达式的逻辑结果为逻辑假时，直接跳过while循环结构，而去执行循环以外的语句。while结构的基本使用方式如示例脚本8-7.sh所示。

```
#示例脚本8-7.sh while结构使用实例
#!/bin/bash

num=1
echo while循环开始
while [ $num -le 6 ]
do
    echo "the current num is $num"
```

```
let num++  
done  
echo while循环结束
```

为示例脚本8-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-7.sh  
ben@ben-laptop:~$ ./8-7.sh  
while循环开始  
the current num is 1  
the current num is 2  
the current num is 3  
the current num is 4  
the current num is 5  
the current num is 6  
while循环结束
```

通过脚本8-7.sh的执行结果可以看出，在使用**while**循环结构时，当满足循环条件时，将会一直执行结构中的语句。当不满足循环条件时，就会自动退出循环，从而执行循环下面的语句。

注意

在使用**while**循环结构时，要注意在计算的过程中，逻辑表达式**expression**的结果要出现逻辑假，否则**while**循环结构会一直执行，形成死循环。

8.2.2 多条件的while结构

在while结构中，expression可以是一个表达式，也可以是多个表达式。对于多个表达式来说，Shell首先依次执行每一个表达式，而循环体是否能够执行，则按照最后一个表达式的结果进行判断。如果最后一个表达式的逻辑运算结果为真，那么while结构的循环体部分就会被执行，如果最后一个表达式的逻辑运算结果为假，即使前面的表达式的逻辑运算结果都为真，循环体部分仍然不会被执行。多条件的while结构如示例脚本8-8.sh所示。

```
#示例脚本8-8.sh 多条件的while结构
#!/bin/bash

echo "使用多个表达式的while结构"
num=4

while [ $num -gt 3 -a $num -lt 6 ]
do
    echo num=$num
    let num++
done
```

为示例脚本8-8.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-8.sh
ben@ben-laptop:~$ ./8-8.sh
使用多个表达式的while结构
```

```
num=4  
num=5  
num=6
```

通过示例脚本8-8.sh的执行结果可以看出，在使用多个表达式时，需要用逻辑连接符将多个表达式进行连接，然后按照逻辑连接符的作用方式对表达式进行计算。最终的结果将作为测试命令的运算结果，从而决定循环结构的执行方式。

8.2.3 使用until命令

until命令的使用方式和while命令类似，也是通过判断逻辑表达式的结果来决定是否执行循环体中的语句。until命令的基本形式如下：

```
until test expression  
do  
    commands;  
done;
```

在上面的循环结构中，首先判断表达式expression的逻辑结果，如果表达式expression结果为逻辑真，即表达式expression的返回状态为0，那么until命令中的commands部分就不会被执行。只有当表达式expression结果为逻辑假，即表达式expression的返回状态为非0的数值时，until命令中的commands部分才会被执行。关于until命令的使用方式如示例脚本8-9.sh所示。

```
#示例脚本8-9.sh  until结构使用实例
#!/bin/bash

num=1
echo until循环开始
until [ $num -ge 6 ]
do
    echo "the current num is $num"
    let num++
done
echo until循环结束
```

为示例脚本8-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-9.sh
ben@ben-laptop:~$ ./8-9.sh
until循环开始
the current num is 1
the current num is 2
the current num is 3
the current num is 4
the current num is 5
the current num is 6
until循环结束
```

通过脚本8-9.sh的执行结果可以看出，在使用until循环结构时，直

到满足循环判断条件，循环才会退出。也就是当num的值大于等于6时，循环退出。当num的值小于6时，一直在循环中运行。

注意

`until`结构是一类单独的结构，但是其使用方式和`while`结构类似，所以在此将`until`结构和`while`放在同一小节中进行讲述。

对于`until`命令来说，逻辑表达式`expression`也可以由多个表达式构成。当`expression`是多个表达式时，决定循环体否被执行的依据也是最后一个表达式的值，而与其他的表达式的运算无关。关于使用多个表达式的`until`命令如示例脚本8-10.sh所示。

```
#示例脚本8-10.sh until结构多条件使用实例
```

```
#!/bin/bash
```

`echo`使用多个表达式的`until`结构

```
num=5
```

```
while [ $num -lt 10 -o $num -gt 5 ]
```

```
do
```

```
echo num=$num
```

```
let num++
```

```
if [ $num -eq 9 ]
```

```
then  
  
    break  
  
fi  
done
```

为示例脚本8-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-10.sh  
ben@ben-laptop:~$ ./8-10.sh  
使用多个表达式的until结构  
num=5  
num=6  
num=7  
num=8  
num=9
```

通过脚本8-10.sh的执行结果可以看出，在使用多个表达式作为判断条件时，需要对循环条件进行仔细的斟酌。

8.3 命令的嵌套

在使用用于循环操作的命令时，循环体部分可以是单独的语句，也可以是其他语句的组合，还可以是其他的循环命令。如在for命令中使用for命令、while命令、until命令等，这种在一个结构中套用该结构的使

用方式称为嵌套。

8.3.1 for命令的嵌套

for命令在嵌套使用时可以嵌套另外一个for命令，从而使得当外层的for命令满足运行条件时，不断地执行内层的for命令。而内层的for命令则按照for命令本身的运行规则运行。常用的for命令嵌套的基本格式如下：

```
for var1 in var_list1
do

    for var2 in var_list2
    do
        commands1;
    done
    commands2
done
```

在上面的结构中，迭代从var_list1中取出第一个值，取出的数值作为变量var1的值，然后执行内层的循环语句。在执行外层for命令的循环语句时，按照for命令的执行方式执行，从var_list2中取值并将取到的值作为var2的数值，进而运行commands1。当从var_list2中取完最后一个值以后，内层的for命令执行完毕，从而继续执行commands2中的语句。这样外层的for命令执行完第一次循环。然后从var_list1中获取第二个数

值，从而再次进入循环体中重新执行内层的for命令以及commands2。上述结构的执行过程如图8-1所示。

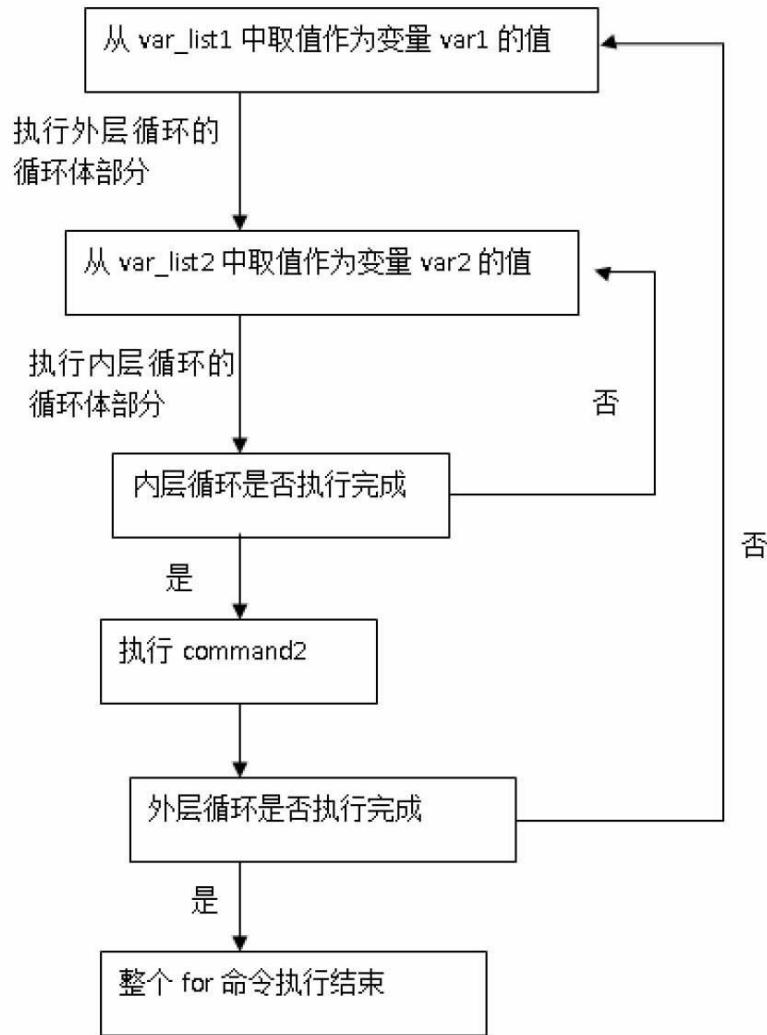


图8-1 for命令嵌套使用的执行过程

使用C式for命令的嵌套方式如下：

```
for (( expression11; expression12; expression13 ))
{
    for (( expression21; expression22; expression23 ))
```

```
{  
    commands2;  
}  
  
commands1;  
}
```

对于C式for命令的循环嵌套使用来说，语句的执行方式和C语言中for循环的执行方式类似。首先执行外层for命令中的表达式expression11来为变量赋初始值，然后执行expression12从而对变量进行逻辑判断，如果逻辑判断的结果为真，那么就执行外层for命令的循环体部分。当进入循环以后，按照C式for命令的执行规则去执行内层的for命令。当内层的for命令执行完毕后，再对外层for命令中的变量值按照expression13为变量重新赋值，然后再判断表达式expression12是否成立。如果expression12的结果为逻辑真，那么继续重新执行内层的for命令以及commands1，直至逻辑表达式expression12得到的结果为逻辑假，整个for命令才算执行完毕，不再执行循环体部分。C式for命令的嵌套结构执行过程如图8-2所示。

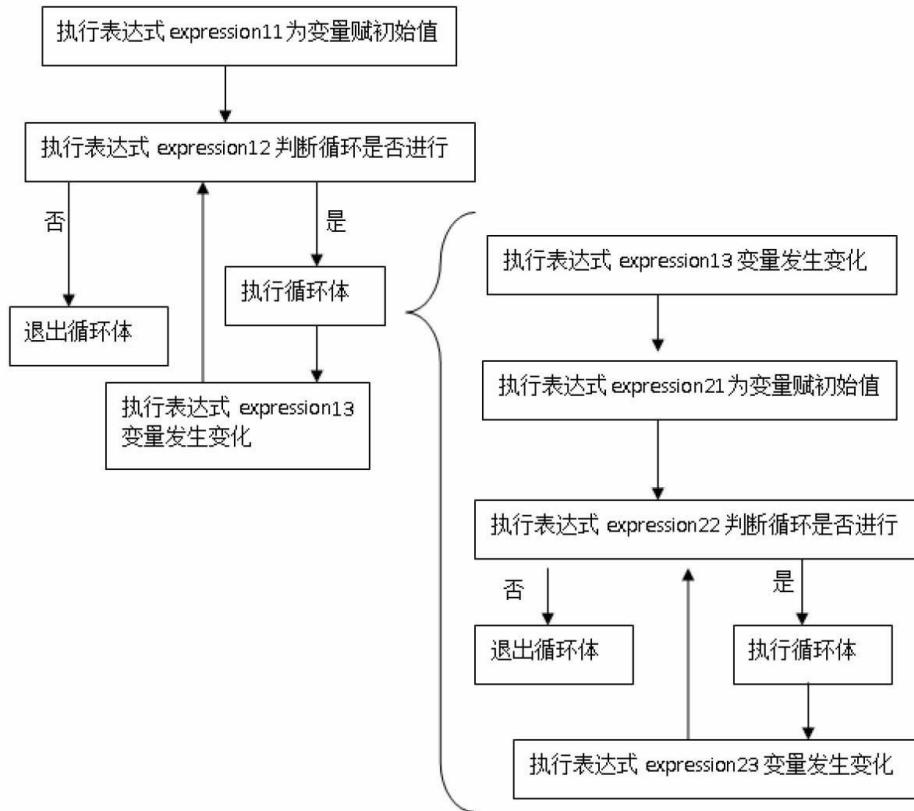


图8-2 C式for命令嵌套执行顺序

for结构的嵌套使用示例如示例脚本8-11.sh所示。

```

#示例脚本8-11.sh  for结构的嵌套使用
#!/bin/bash

echo "for命令的嵌套使用"

num1=1;
num2=2;

for (( num1=1; num1<3; num1++ ))
{

```

```
echo "外层循环体中， 变量num1的值为: $num1";  
  
for (( num2=1; num2<5; num2++ ))  
  
{  
  
    echo "内层循环体中， 变量num2的值为: $num2";  
  
}  
echo  
}
```

为示例脚本8-11.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-11.sh  
ben@ben-laptop:~$ ./8-11.sh  
for命令的嵌套使用  
外层循环体中， 变量num1的值为: 1  
内层循环体中， 变量num2的值为: 1  
内层循环体中， 变量num2的值为: 2  
内层循环体中， 变量num2的值为: 3  
内层循环体中， 变量num2的值为: 4  
  
外层循环体中， 变量num1的值为: 2
```

```
内层循环体中，变量num2的值为：1  
内层循环体中，变量num2的值为：2  
内层循环体中，变量num2的值为：3  
内层循环体中，变量num2的值为：4
```

通过示例脚本8-11.sh的执行结果可以看出，在使用嵌套循环时，首先进入外层循环，在遇到新的循环体之后，需要将新的循环体执行完毕，才能执行外层循环的其他内容，直至执行完毕。而当再次执行外层循环的内容时，内层循环还会从头开始执行一次。

8.3.2 while命令的嵌套

在Shell脚本中，while命令和until命令都可以使用嵌套结构。在使用嵌套结构时，可以是while命令之间的嵌套或until命令之间的嵌套，也可以是while命令和until命令之间的嵌套。在本小节中仅介绍while命令之间的嵌套和until命令之间的嵌套。至于until命令之间的嵌套，因为它和while命令之间的嵌套基本一致，所以在此不再专门讲述。

嵌套使用while命令的基本结构如下：

```
while test expression1  
do  
    while test expression2  
    do  
        commands2;  
    done
```

```
commands1;  
done
```

while命令嵌套使用时也是首先判断expression1的逻辑运算结果，如果运算结果为逻辑真，那么就进入循环体内部执行内层的while命令和commands1，当执行完以后，再次判断表达式expression1是否成立。只有当表达式的逻辑运算结果为假时，内层的while命令和commands1才不会被执行。每次执行内层循环时，while命令都会重新执行。关于while命令嵌套使用的实例如示例脚本8-12.sh所示。

```
#示例脚本8-12.sh  while结构嵌套使用实例  
#!/bin/bash  
  
num=1  
echo while循环开始  
while [ $num -le 3 ]  
do  
    num2=1  
    while [ $num2 -le 3 ]  
    do  
        echo num2 = $num2  
        let num2++  
    done  
    echo 内层循环结束  
    echo the current num is $num
```

```
let num++  
echo  
done  
echo while循环结束
```

为示例脚本8-12.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-12.sh  
ben@ben-laptop:~$ ./8-12.sh  
while循环开始  
num2 = 1  
num2 = 2  
num2 = 3  
内层循环结束  
the current num is 1  
  
num2 = 1  
num2 = 2  
num2 = 3  
内层循环结束  
the current num is 2  
  
num2 = 1  
num2 = 2  
num2 = 3  
内层循环结束
```

```
the current num is 3
```

```
while循环结束
```

通过示例脚本8-12.sh的执行结果可以看出，使用while嵌套循环时，内层循环每一次都会重新执行，内层循环和外层循环互不干涉。

while命令的嵌套除了可以嵌套使用while命令以外，还可以嵌套使用until结构以及其他其他的结构。在嵌套使用其他结构时，要特别注意各种结构命令的可执行条件以及执行过程，既要防止出现死循环，又要防止出现循环不运行的情况。

8.3.3 until命令的嵌套

until命令也可以实现嵌套使用，在进行嵌套使用时，可以嵌套使用until结构，还可以嵌套使用while结构、for结构以及其他其他的语句，可以根据实际需要选择使用哪种结构。until结构的使用实例如示例脚本8-13.sh所示。

```
#示例脚本8-13.sh until结构嵌套while结构使用实例
```

```
#!/bin/bash
```

```
num=1
```

```
echo until循环开始
```

```
until [ $num -ge 6 ]
```

```
do
```

```
num2=1

while [ $num2 -le 3 ]

do

    echo num2 = $num2

    let num2++

done

echo 内层while循环结束

echo the current num is $num

let num++

echo

done

echo until循环结束
```

为示例脚本8-13.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-13.sh
ben@ben-laptop:~$ ./8-13.sh
until循环开始
num2 = 1
num2 = 2
num2 = 3
内层while循环结束
```

```
the current num is 1
```

```
num2 = 1
```

```
num2 = 2
```

```
num2 = 3
```

内层while循环结束

```
the current num is 2
```

```
num2 = 1
```

```
num2 = 2
```

```
num2 = 3
```

内层while循环结束

```
the current num is 3
```

until循环结束

通过示例脚本8-13.sh的执行结果可以看出，通过until嵌套使用其他的结构时，同样要注意循环条件的使用，否则会造成执行顺序混乱的情况。

注意

在使用循环结构时，要特别注意循环条件的使用，防止死循环或不循环的现象发生。

8.4 循环控制符

在使用循环命令时，并非只有当相应的逻辑表达式计算结果为假时，循环才会退出。在Shell脚本中，还可以使用break命令和continue命令来中断当前循环命令的运行，转而运行循环体外的内容或其他相应的内容。本小节重点讲述如何使用break命令和continue命令来控制循环的运行。

8.4.1 使用break中断

break命令的使用方式和其他高级编程语言中的使用方式类似，都是跳出本循环，转而执行循环外部的语句。在Shell脚本中，break还有自己独特的用法，就是可以选择跳出几层循环。在Shell中，break的基本使用方式如下：

```
break n;
```

在上面的结构中，n代表break命令将要跳出的层数，而n一般是从2开始取值，并且只能取正整数。当n为1时，break只能跳出本层循环，而此时一般省略不写。下面首先看一下break命令的基本使用，如示例脚本8-14.sh所示。

```
#示例脚本8-14.sh break的简单使用
#!/bin/bash
```

```
echo "使用break命令中断for循环执行"
for (( i = 1; i <= 10; i++ ))
do
    echo "当前的变量值为: $i";
    if [ $i -eq 6 ]
    then
        echo "跳出循环之前变量值为: $i";
        break;
    fi
done
```

为示例脚本8-14.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-14.sh
ben@ben-laptop:~$ ./8-14.sh
使用break命令中断for循环执行
当前的变量值为: 1
当前的变量值为: 2
当前的变量值为: 3
当前的变量值为: 4
当前的变量值为: 5
当前的变量值为: 6
跳出循环之前变量值为: 6
```

break命令不但能够跳出for命令组成的循环，还能跳出while命令和until命令组成的循环。而在使用嵌套的循环中，特别需要注意break命令

到底是跳出本层循环，还是跳出第几层循环。关于break的复杂使用如示例脚本8-15.sh所示。

```
#示例脚本8-15.sh break的复杂使用
#!/bin/bash

echo "使用break命令中断for循环执行"
for (( num1 = 1; num1 >= 10; num1++ ))
do

    for (( num2 = 1; num2 <= 10; num2++ ))
do

    echo "变量num1的当前值为: $num1";
    echo "变量num2的当前值为: $num2";

    if [ $num2 -eq 5 ]
    then
        echo退出内层循环后继续外层循环
        break;
    fi;

    if [ $num1 -eq 3 ]
    then
        echo 退出2层循环
        break 2;
    fi
done done
```

echo结束整个循环

为示例脚本8-15.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-15.sh
ben@ben-laptop:~$ ./8-15.sh
使用break命令中断for循环执行
变量num1的当前值为: 1
变量num2的当前值为: 1
变量num1的当前值为: 1
变量num2的当前值为: 2
变量num1的当前值为: 1
变量num2的当前值为: 3
变量num1的当前值为: 1
变量num2的当前值为: 4
变量num1的当前值为: 1
变量num2的当前值为: 5
退出内层循环后继续外层循环
变量num1的当前值为: 2
变量num2的当前值为: 1
变量num1的当前值为: 2
变量num2的当前值为: 2
变量num1的当前值为: 2
变量num2的当前值为: 3
变量num1的当前值为: 2
变量num2的当前值为: 4
```

```
变量num1的当前值为: 2
变量num2的当前值为: 5
退出内层循环后继续外层循环
变量num1的当前值为: 3
变量num2的当前值为: 1
退出2层循环
结束整个循环
```

通过示例脚本8-15.sh的运行结果可以看出，使用break时，默认跳出本层循环，还可以指定跳出几层循环，从而使得脚本的执行顺序更加灵活。

注意

break命令跳出循环后，其后面的语句以及未完成的循环将不再执行。但是循环之外的语句正常执行。

8.4.2 使用**continue**继续

continue命令也可以用于改变循环结构的执行过程，但是continue命令不像break命令那样直接跳出循环，而是结束本次循环，从而直接运行下一次循环。continue命令的基本使用方式如示例脚本8-16.sh所示。

```
#示例脚本8-16.sh  continue命令的使用
#!/bin/bash
```

```
echo "使用continue命令中断本次循环执行"
for (( i = 1; i <= 10; i++ ))
do
if [ $i -eq 6 ]
then
echo "跳出循环之前变量值为: $i";
continue;
fi
echo "当前的变量值为: $i";
done
```

为示例脚本8-16.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 8-16.sh
ben@ben-laptop:~$ ./8-16.sh
使用continue命令中断本次循环执行
当前的变量值为: 1
当前的变量值为: 2
当前的变量值为: 3
```

```
当前的变量值为: 4
当前的变量值为: 5
跳出循环之前变量值为: 6
当前的变量值为: 7
当前的变量值为: 8
当前的变量值为: 9
当前的变量值为: 10
```

通过示例脚本8-16.sh的运行结果可以看出，在未执行continue命令时，在循环的最后都会输出一条语句，显示变量当前的数值是多少。当执行了continue命令后，输出变量值的语句不再执行，转而执行新一轮的循环。

在使用continue命令时，特别需要注意的是当执行了continue命令以后，该命令后面的语句将不再执行，转而执行下一次循环。因此要执行的语句需要放在continue语句之前，防止发生应该执行的语句未执行的情形。

8.5 小结

本章重点介绍了循环命令在Shell脚本中的使用方式。在编写Shell脚本时，可以使用for命令、while命令、until命令以及这3个命令的相互嵌套组成的循环结构。还可以根据实际需要，使用break命令和continue命令来改变循环的执行过程。

for命令可以迭代变量列表，无论是以命令行形式提供的还是用变量名提供的，还可以通过通配符获取文件或目录的名称。

`while`命令则是通过判断一个逻辑表达式的结果来决定是否执行循环部分。只有当表达式的结果为逻辑真时，循环部分才会被执行。

`until`命令的执行方式和`while`命令相反，当表达式的返回结果为非0值时（逻辑假），循环部分才会被执行。

在循环执行的过程中还可以使用`break`命令或`continue`命令中断循环的执行。`break`命令用来跳出本层循环或跳出几层循环，可以使用参数进行确定。`continue`命令用来结束本次循环，转而执行下一次循环。在执行了`break`语句以后，`break`以后的语句以及未完成的循环都不再执行，而执行了`continue`语句之后，`continue`语句后面的部分将不再执行。但是下一次循环执行时，如果`continue`没有执行，后面的语句还会被执行。

第9章 Shell中的函数

在执行Shell命令时，有时候经常执行某些语句，但是这些语句不会循环执行。此时如果需要的语句很少的话，可以在需要执行语句的地方添加相应的语句。而当语句很多，书写量比较大时，则需要使用另外以一种方式，这种方式就是函数。本章将重点介绍Shell脚本中函数的使用方式。

本章的主要内容如下：

- 函数的基本用法
- 函数的返回值
- 函数中全局变量和局部变量的区别，以及数组在函数中的作用
- 函数的递归使用
- 函数的嵌套使用

9.1 函数的基本用法

在使用高级语言（如C语言）编写较复杂的程序时，一般采用模块化的设计方式将系统分成一个一个细小的模块，每一个模块一般通过一个函数实现。在需要调用函数的时候，可以使用函数名代替整个函数体，从而完成函数体中语句的调用。在Shell脚本中，函数的作用也是将一些语句集合在一起，在需要调用这些语句时，使用函数名来代替。

9.1.1 函数的创建与使用

在Shell脚本中，使用函数之前首先要创建函数。创建函数一般有两种方式，第1种方式如下所示：

```
function name
{
    commands;
}
```

在上面的基本结构，使用命令function定义了名字为name的函数，在函数中执行的语句是commands。函数名称name在同一个脚本文件中必须是唯一的，否则在执行函数时，bash将不知道需要执行哪一个函数。commands由一条或多条语句组成，在函数调用时，bash会按照语句的先后顺序去执行函数体中的每一条语句。

在Shell脚本中，还存在另外一种创建函数的形式，其基本结构如下所示：

```
name()
{
    commands;
}
```

在上面的结构中，省略了命令function，而使用一对小括号“()”来代替命令function，从而让bash知道name是一个函数。而函数体部分仍然

需要按照语句的先后顺序逐条执行。

函数创建以后可以在当前脚本文件中适合的地方进行调用。在Shell脚本中，调用函数的方式和调用普通bash命令的方式类似，直接使用函数名即可实现函数的调用。关于函数的基本创建方式如示例脚本9-1.sh所示。

```
#示例脚本9-1.sh 创建函数的基本方式
#!/bin/bash

echo "在shell脚本中使用函数"
echo "使用命令function创建函数"
function fun1
{
    echo "使用命令function创建函数";
}

echo "调用函数fun1"
fun1

echo "不使用命令function创建函数"
function fun2
{
    echo "在函数fun2中";
}

echo "调用函数fun2"
```

fun2

为示例脚本9-1.sh赋予可执行权限后执行脚本，结果如下所示：

```
ben@ben-laptop:~$ chmod u+x 9-1.sh
ben@ben-laptop:~$ ./9-1.sh
在shell脚本中使用函数
使用命令function创建函数
调用函数fun1
使用命令function创建函数
不使用命令function创建函数
调用函数fun2
在函数fun2中
```

在使用函数时，要注意函数的调用要在函数的创建之后，也就是说在使用函数之前，必须要创建函数，如果调用未创建的函数，那么在执行函数的过程中会发生错误，如示例脚本9-2.sh所示。

```
#示例脚本9-2.sh 使用未创建的函数
#!/bin/bash

echo "在shell脚本中使用未创建的函数fun1"

fun1

function fun1
```

```
{  
    echo "在调用函数fun1之后创建函数";  
}
```

为示例脚本9-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-2.sh  
ben@ben-laptop:~$ ./9-2.sh  
  
“在shell脚本中使用未创建的函数fun1”  
ben@ben-laptop:~$ ./9-2.sh: line 4: fun1: 找不到命令
```

通过示例脚本9-2.sh的执行结果可以看出，若使用未创建的函数，在函数执行时将会提示“找不到命令”。因此如果想使用某个函数，首先需要创建函数，然后才可以使用该函数。

9.1.2 函数的参数

在其他的编程语言中，在使用函数时可以给函数添加参数，这些参数一般是在函数运行时才会确定具体的数值。在Shell脚本中，函数也可以使用参数。但是这些参数必须是由标准参数变量表示并且由命令行传递给函数的参数。这些参数一般包括\$0、\$1、\$2……\$#等。其中\$0一般用来表示函数名，而\$1、\$2等用来表示传递给函数的第1个、第2个参数等。\$#一般用来表示传递给函数的参数的个数。关于函数参数的使用方式如示例脚本9-3.sh所示。

```
#示例脚本9-3.sh 使用函数参数

#!/bin/bash

echo "在函数fun中使用函数参数"

fun()
{
    echo "传入的参数的个数为: $#";
    echo "函数名是: $0";
    echo "第1个参数是: $1";
    echo "第2个参数是: $2";
    echo "第3个参数是: $3";
    echo "第4个参数是: $4";
    echo "第5个参数是: $5";

}

echo "调用函数, 并传递5个参数"
fun 1 2 3 4 5
fun "hello" "world" "is" "the" "first" "shell script"
```

为示例脚本9-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-3.sh
ben@ben-laptop:~$ ./9-3.sh
在函数fun中使用函数参数
调用函数, 并传递5个参数
```

```
传入的参数的个数为: 5
```

```
函数名是: 9-3.sh
```

```
第1个参数是: 1
```

```
第2个参数是: 2
```

```
第3个参数是: 3
```

```
第4个参数是: 4
```

```
第5个参数是: 5
```

```
传入的参数的个数为: 6
```

```
函数名是: ./9-3.sh
```

```
第1个参数是: hello
```

```
第2个参数是: world
```

```
第3个参数是: is
```

```
第4个参数是: the
```

```
第5个参数是: first
```

通过示例脚本9-3.sh的执行结果可以看出，在函数中使用参数，可以使用符号\$1~\$9来代替传递给函数的参数，并且使用\$0来表示函数名，这样在编写脚本时，可以非常方便地对传递的参数进行各种操作。

注意

\$#表示的参数的个数不包括函数名。函数名不在函数参数的范围之列。

9.2 函数的返回值

在高级编程语言（如C语言）中，函数在运行完之后，都会返回一个相应的数值，这个数值用来判断函数的执行结果。而在Shell脚本中，不但函数同样会在执行完之后返回一个数值，即使普通的命令在执行完毕后都会产生返回值。本小节将首先介绍什么是返回值，然后重点介绍函数的返回值。

9.2.1 返回值基础

在Bash Shell中，任何命令执行完成后都会产生一个返回值，这个返回值用来判断命令执行是否正确。在Unix系统中，一般使用0表示命令执行成功，而非0值表示执行失败，不同的数值表示不同的错误方式。要想获取命令的返回值，一般使用参数“\$?”，该参数自动记录上一个命令的返回值。关于参数“\$?”的使用方式如示例脚本9-4.sh所示。

```
#示例脚本9-4.sh 使用函数返回值
#!/bin/bash

echo "返回普通命令的返回值"
ls /home
echo "正确执行命令ls后返回值为: $?"
ls /no_exist
echo "错误执行命令ls后返回值为: $?"

ls /no_exist
ls /home
echo "上一条命令是正确执行命令ls后返回值为: $?"
```

```
ls /home  
ls /no_exist  
echo “上一条命令是错误执行命令ls后返回值为: $?”
```

为示例脚本9-4.sh赋予可执行权限后执行脚本，结果如下所示：

```
ben@ben-laptop:~$ chmod u+x 9-4.sh  
ben@ben-laptop:~$ ./9-4.sh  
返回普通命令的返回值  
ben  
正确执行命令ls后返回值为: 0  
ls: 无法访问/no_exist: 没有那个文件或目录  
错误执行命令ls后返回值为: 2  
上一条命令是正确执行命令ls后返回值为: 0  
上一条命令是错误执行命令ls后返回值为: 2
```

通过示例脚本9-4的执行结果可以看出，符号“\$?”自动记录最后一个Shell命令的返回值，而其他命令的返回值被下一条命令的返回值所覆盖。因此，如果需要使用某个命令的返回值，就需要及时使用，防止被覆盖。

9.2.2 函数的默认返回值

在默认的情况下，函数的返回值是最后一条命令的返回值，当函数

执行完毕后，会将最后一条命令的返回值作为整个函数的返回值，可以通过获取参数“\$?”的值来获取函数的返回值，如示例脚本9-5.sh所示。

```
#示例脚本9-5.sh 函数默认返回值
#!/bin/bash

echo "获取函数的默认返回值"
fun1()
{
    echo "最后一条命令为执行正确的命令"
    echo "hello world"
}

fun1
echo "函数fun1的返回值为: $?"
echo

fun2()
{
    echo "最后一条命令为执行错误的命令"
    ls /no_exist.sh
}

fun2
echo "函数fun2的返回值为: $?"
```

为示例脚本9-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-5.sh
```

```
ben@ben-laptop:~$ ./9-5.sh
```

获取函数的默认返回值

最后一条命令为执行正确的命令

```
hello world
```

函数fun1的返回值为：0

最后一条命令为执行错误的命令

```
ls: 无法访问/no_exist.sh: 没有那个文件或目录
```

函数fun2的返回值为：2

通过示例脚本9-5.sh的执行结果可以看出，在默认的情况下，函数的返回值是函数中最后一条命令的返回值，而与其他的命令执行结果无关。

注意

函数的默认返回值取决于最后一个命令的执行结果，因此一般不使用默认返回值作为函数的返回值。

9.2.3 return命令的使用

在Shell脚本中，函数的返回值除了使用默认返回值以外，一般使用

`return`命令来返回特定的返回值，从而使得函数能够按照特定的状态返回。使用`return`命令的基本方式如下所示：

```
return value;
```

在使用`return`语句时，`value`是函数的返回值，这个返回值必须在0~255之间。如果`value`值大于255，`bash`会使用其他合适的值代替。`return`命令的基本使用方式如示例脚本9-6.sh所示。

```
#示例脚本9-6.sh 使用return返回特定返回值
#!/bin/bash

echo "使用return命令返回函数的返回值"
fun1()
{
    for (( num=1; num<10; num++ ))
    {
        if [ $num -eq 5 ]
        then
            return $num
        fi
    }
}
fun1
```

```
echo "函数fun1的返回值为: $?"  
echo  
  
fun2()  
{  
  
    echo "使用return命令返回大于255的数值"  
    return 256  
  
}  
fun2  
echo "函数fun2的返回值为: $?"
```

为示例脚本9-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-6.sh  
ben@ben-laptop:~$ ./9-6.sh  
使用return命令返回函数的返回值  
函数fun1的返回值: 5  
  
使用return命令返回大于255的数值  
函数fun2的返回值: 0
```

通过示例脚本9-6.sh的执行结果可以看出，使用return语句可以为函数返回一个特定的返回值，如果该数值超过255，那么就按照0返回，而不再返回特定的数值。

注意

在使用**return**命令返回函数的返回值时，要尽快地取值，否则会被后续执行的命令的返回值覆盖。

9.2.4 使用函数的返回值

函数的返回值除了使用参数“\$?”显示以外，还可以直接将函数的返回值赋给某个变量，在使用这种方式时，函数可以返回任意类型的数值给变量，如下所示：

```
var='function'
```

上面的表达式的作用是将函数**function**的执行结果赋值给变量**var**，变量**var**可以接受任何类型的函数返回值。使用函数返回值的实例如示例脚本9-7.sh所示。

```
#示例脚本9-7.sh 使用函数的返回值
#!/bin/bash

echo “使用return命令返回函数的返回值”
fun1()
{
for (( num=1; num<10; num++ ))

```

```
{  
  
    if  [$num -eq 5]  
        then  
            return $num  
    fi  
}  
}  
  
fun1  
  
echo “函数fun1的返回值为: $?”
```

为示例脚本9-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-7.sh  
ben@ben-laptop:~$ ./9-7.sh  
使用return命令返回函数的返回值  
函数fun1的返回值为： 5
```

通过示例脚本9-7.sh的执行结果可以看出，使用return命令可以返回特定的返回值，而使用符号“\$?”可以来接收函数的返回值。但是return命令的返回值只能在0到255之间，否则就会按照0返回。

9.3 函数中的全局变量和局部变量

在其他的编程语言中，变量用来存储程序运行过程中不断变化的内容，而在Shell脚本中，仍然需要使用变量来表示一些不确定的数值。根

据变量作用域的不同可以将变量分为以下两类：

- 全局变量
- 局部变量

本节就重点介绍全局变量和局部变量在函数中的具体使用。

9.3.1 全局变量

全局变量是在整个脚本文件中都有效的变量，全局变量可以在函数外定义，也可以在函数内部定义。不管在什么地方定义了全局变量，只要是定义了该变量，在Shell脚本的任何地方都可以使用该变量。全局变量在Shell中的使用如示例脚本9-8.sh所示。

```
#示例脚本9-8.sh 全局变量的使用
#!/bin/bash

echo "在函数中使用全局变量"
num=10
string="hello world"
fun1()
{
    $num=$[ $num + 10 ]
    echo "全局变量string = $string"
}
fun1
```

```
echo "在函数fun1中变化以后，变量num的值为: $num"
```

为示例脚本9-8.sh赋予可执行权限后执行脚本，结果如下所示：

```
ben@ben-laptop:~$ chmod u+x 9-8.sh
ben@ben-laptop:~$ ./9-8.sh
在函数中使用全局变量
全局变量string = hello world
在函数fun1中变化以后，变量num的值为: 20
```

通过示例脚本9-8.sh的执行结果可以看出，全局变量可以在全局使用，其作用范围为定义之后直到脚本文件结束。

注意

变量在定义时默认是全局变量。因此在使用全局变量时，特别需要注意如果多个函数调用同一个变量，需要明确该全局变量在函数中发生了何种变化，以防止误操作。

9.3.2 局部变量

局部变量，顾名思义就是变量的作用域局限在某个地方，而不能随意应用。就像是某些图书馆中的图书只能在阅览室中阅读，而不能拿到阅览室之外的任何地方阅读。在定义局部变量时需要使用命令local，定义局部变量的一般方式如下所示：

```
local valnum
```

在上面的结构中，valnum可以是单个的变量，也可以是一个赋值语句。在使用local命令限定变量为局部变量以后，该变量只能在函数内使用。而由于变量的作用域只局限于函数内部，所以如果在函数外部或其他地方定义了同名的全局变量，bash会很好地处理两个变量的关系，使得这两个变量都能正常使用。局部变量在函数中的使用实例如示例脚本9-9.sh所示。

```
#示例脚本9-9.sh 局部变量的使用
#!/bin/bash

echo "在函数中使用局部变量"
num=10
string="hello world"
fun1()
{
    local num=100
    local string="local value"
    echo "在函数fun1中调用获取的变量num的值为: $num"
    echo "在函数fun1中调用获取的变量string = $string"
    num=$(( $num + 100 ))
}
fun1
```

```
echo "在函数fun1外， 变量num的值为 : $num"  
echo "在函数fun1外， 变量string的值为 : $string"
```

为示例脚本9-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-9.sh  
ben@ben-laptop:~$ ./9-9.sh  
在函数中使用局部变量  
在函数fun1中调用获取的变量num的值为: 100  
在函数fun1中调用获取的变量string = local value  
在函数fun1外， 变量num的值为 : 10  
在函数fun1外， 变量string的值为 : hello world
```

通过示例脚本9-9.sh的执行结果可以看出，局部变量只能在当前函数中有效，而在其他的地方则不会生效，也就无法使用。

9.4 数组与函数

在Shell脚本中，数组是一类比较特殊的变量，使用数组可以同时表示多个变量。数组可以用在Shell脚本的任何地方，函数也不例外。但是在函数中使用数组却存在诸多的限制。本节就重点介绍数组在函数中的使用方式。

9.4.1 数组作为函数参数

函数在运行时可以使用由标准参数变量表示并且由命令行传递给函数的参数，这些参数的类型可以是任意的类型，同样数组类型的变量也可以作为参数传递给函数，但是不能将数组作为一个整体传递给函数，否则函数只能提取数组中的第一个数值，并且提示“数组错误”。

为了使数组能够作为函数的参数并且保证脚本能够正确地执行，需要将数组中的每一个数据单独传递给函数，并且在函数内部使用局部数组变量重新将传递的参数整合成数组，然后在函数中使用数组。数组作为函数的参数的使用实例如示例脚本9-10.sh所示。

```
#示例脚本9-10.sh 数组作为函数参数
#!/bin/bash

echo "正常方式下使用数组作为函数的参数"
num1=(1,2,3,4,5)
num2=(a,b,c,d,e)

fun1()
{
    local array1=$1
    echo "传递的数组1中的数值分别为: ${array1[*]}"

    local array2=$2
    echo "传递的数组2中的数值分别为: ${array2[*]}"
}

}
```

```
fun1 "${num1[*]}"  
echo  
  
echo "向函数传递多个数组作为参数"  
fun2 "${num1[*]}""${num2[*]}"
```

为示例脚本9-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-10.sh  
ben@ben-laptop:~$ ./9-10.sh  
正常方式下使用数组作为函数的参数  
传递的数组1中的数值分别为  
1,2,3,4,5  
传递的数组2中的数值分别为：  
  
向函数传递多个数组作为参数  
传递的数组1中的数值分别为  
1,2,3,4,5  
传递的数组2中的数值分别为：  
a,b,c,d,e
```

通过示例脚本9-10.sh的执行结果可以看出，当函数运行时，可以传递数组作为函数的参数。当传递的参数中包含多个数组时，可使用变量 \$1、\$2等进行区分，从而使函数能够使用正确的参数进行数据的处理。

9.4.2 数组作为函数返回值

使用return命令可以将函数中任意类型的数值作为函数的返回值，同样，数组也可以作为函数的返回值。当数组作为函数的返回值时，需要使用另外一个数组接收函数的返回值。数组作为函数返回值的使用实例例如示例脚本9-11.sh所示。

```
#示例脚本9-11.sh 数组作为函数返回值
#!/bin/bash

fun()
{
    local old
    local new
    local num
    local i
    old=(`echo "$@"`)
    new=(`echo "$@"`)
    num=${# - 1}
    echo "传递给函数的数组为: ${old[*]}"

    for (( i = 0; i <= $num; i++ ))
    {
        new[$i]=${old[$i]} * 2
    }
    echo ${new[*]}
}

array=(1 2 3 4 5)
echo "数组的值为: ${array[*]}"
arg1=`echo ${array[*]}`
```

```
result=(`fun $arg1`)
echo "函数的返回数组为: ${result[*]}"
```

为示例脚本9-11.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-11.sh
ben@ben-laptop:~$ ./9-11.sh
数组的值为 1 2 3 4 5
传递给函数的数组为: 1 2 3 4 5
函数的返回数组为: 2 4 6 8 10
```

通过示例脚本9-11.sh的执行结果可以看出，使用echo命令可以将数组作为函数的返回值返回，而函数的返回值也需要使用另外一个数组接受，从而保证数据的一致性。

注意

普通变量不能用来接收数组返回值。

9.5 脚本函数递归

在高级编程语言中，函数可以实现递归调用。在Shell脚本语言中，函数也可以实现递归调用。在递归调用时，可使用递归函数来调用函数本身，直至不再调用函数本身为止。调用过程如图9-1所示。

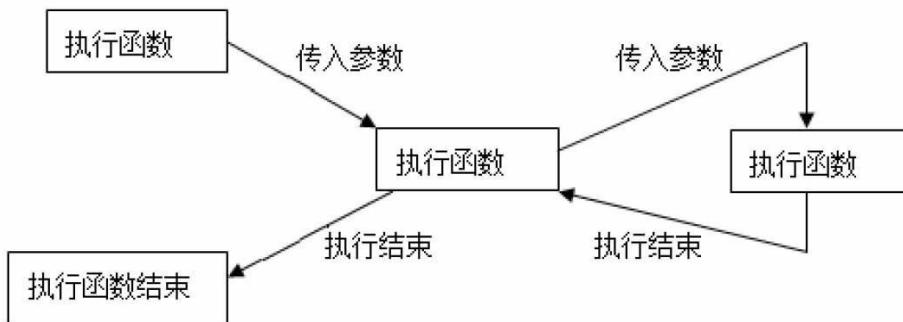


图9-1 递归函数执行过程示意图

函数的递归调用过程需要有一个参数进行控制，这样可以限制调用的次数，否则就会使得函数重复调用，形成死循环。

函数的递归调用一般使用阶乘来作为实例进行讲述，阶乘的Shell版如示例脚本9-12.sh所示。

```

#示例脚本9-12.sh 递归函数的使用
#!/bin/bash

factor()
{
rest=0 fact=0
if (( $1 <= 1));
then
rest=1
return 0
else
((fact = $1 -1 ))
factor $fact      #递归调用函数factor

```

```
((rest=$1*$rest))

rest=$rest

return 0

fi

}

factor $1

echo factor of $1 is $rest
```

为示例脚本9-12.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-12.sh
ben@ben-laptop:~$ ./9-12.sh 5
factor of 5 is 120
```

通过示例脚本9-12.sh的执行结果可以看出，递归函数factor的调用在传递的参数为1时，退出整个函数的调用，当满足其他条件时，不断地调用函数本身来执行。

注意

在使用递归函数时，要注意设定函数退出的条件，否则函数会不停地调用本身，形成“死循环”。

9.6 函数的嵌套调用

函数除了进行递归调用以外，还可以进行嵌套使用。所谓函数的嵌套使用，就是在一个函数中调用另外一个函数。调用函数的方式和普通函数的执行方式类似，函数嵌套调用的基本使用如下所示：

```
fun1()
{
    commands; }

fun2()
{
    fun1
    commands
}
```

在函数fun2()的调用过程中，当执行到函数fun1()时，会根据函数名直接跳转到函数fun1()中去执行，当执行完函数fun1()以后，再执行函数fun2()中的其他内容。函数嵌套使用实例如示例脚本9-13.sh所示。

```
#示例脚本9-13.sh 嵌套函数的使用

#!/bin/bash
echo "函数的嵌套使用实例"
fun1()
{
    echo "在函数fun1\(\)中"
}
fun2()
```

```
{  
    echo "在函数fun2()\\"中"  
    echo  
  
    echo "执行函数fun1()\\"  
    fun1  
  
}  
  
echo "调用函数fun2()\\"  
fun2
```

为示例脚本9-13.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 9-13.sh  
ben@ben-laptop:~$ ./9-13.sh  
函数的嵌套使用实例  
调用函数fun2()  
在函数fun2()中  
  
执行函数fun1()  
在函数fun1()中
```

通过示例脚本9-13.sh的执行结果可以看出，当嵌套调用函数时，首先执行嵌套函数内部的语句，当嵌套函数内部的语句执行完以后，自动跳出嵌套函数，从而继续执行嵌套函数外面的语句，其执行方式如图9-

2所示。

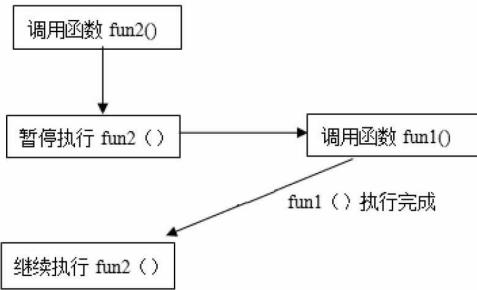


图9-2 函数嵌套执行过程

9.7 小结

在本章中主要介绍了Shell脚本中函数的使用。可以将某些需要重复执行的命令使用函数的方式进行封装，而当需要执行这些命令时，bash执行到函数名，便直接跳转到函数中来执行相应的命令。

创建函数时可以使用关键字function或省略关键字而使用普通函数的创建方式。函数只能在创建之后被执行。函数在执行时，可以使用来自标准参数变量表示并且由命令行传递给函数的参数，从而使得函数的执行更加灵活。

函数在执行过程中，可以使用默认返回值，还可以使用return命令返回特定的返回值。默认的返回值就是函数中最后一条命令的返回值。而return命令可以返回任意类型的返回值。函数的返回值可以赋值给某个变量，从而将函数的返回值添加到主程序中执行。

函数在执行过程中可以使用全局变量和局部变量以及数组变量等各种类型的变量。全局变量是在整个脚本文件中都可以使用的变量，而局部变量只能在函数内部使用。数组变量可以通过特定的方式，作为函数

的参数参与函数的执行。而函数在执行完成后，还可以将数组作为返回值来返回给主程序。

在Shell脚本中，函数可以实现递归调用和嵌套调用。递归调用就是函数不断地调用函数本身，当满足某个条件以后，结束递归的调用过程。如果没有条件使得递归结束，就会形成死循环。函数的嵌套调用就是在在一个函数中调用另外一个函数，当函数执行到另外一个函数时，bash会根据函数名跳转到被调用的函数中去执行相应的语句。

第10章 Shell脚本编写技巧

前面的章节中介绍了Shell脚本编写的大部分知识，至此读者可以根据前面的知识来编写脚本，从而实现需要的功能。但是在使用的过程中，有些技巧还是需要广大读者注意的，如脚本的编写规范以及使用技巧，在本章就重点介绍这部分内容。在Shell脚本中所有的变量都是按照字符串来存储的，在进行算术运算时会出现“特别的结果”，因此本章介绍了在Shell脚本中如何进行算术运算。

本章的主要内容如下：

- 脚本编写规范
- 脚本的优化
- 脚本的使用技巧

10.1 脚本编写规范

编写Shell脚本的最终目的是实现某个特定的功能，为了这个目的可以进行各种“合理合法”的操作，合理就是指脚本编写好以后能够实现最终的功能，并且能够方便后期的维护。而合法是指符合Shell脚本编写的语法知识，从而使得BASH能够顺利解释执行脚本中的命令。脚本在编写的过程中要遵守一定的规范，既能使得脚本能够顺利运行，也使得后期的维护变得非常简单。

10.1.1 命名规范

在编写脚本时，比较重要的是对需要命名的地方使用准确而简单的名字，在编写Shell脚本时，需要命令的地方如下：

- 脚本名称
- 变量名
- 函数名

这3种名字可以参照匈牙利命令法来进行命名。该命名法是现在大部分编程语言中普遍采用的命名方式，其一般的规则就是“见名知意”，也就是说在看到脚本的名称之后，就可以大体知道脚本的作用，而变量名则体现了变量的作用范围、功能等。对于函数名来说，根据函数名就可以大体判断函数的作用。使用此种方式进行命名，即使同时存在很多的脚本文件，也能很容易地找到需要的脚本。下面将简单介绍Shell脚本中匈牙利命名法的使用方式。

(1) 变量名称由字母(a~z和A~Z)和数字组成，一般不使用下划线。开头字符可以是任意的字符。字符大小写在Shell脚本中是敏感的，变量名A和a分别代表两个变量。如下面均是合法命名方式：

```
num=10  
Num="123"  
NUM=hello
```

在上面的语句中，分别使用了变量num来表示整数123，而使用Num和NUM来表示字符串123和字符串hello。但是在实际的应用中应尽量减少使用字符的大小写、字符加数字的方式来区别不同的变量。如下面的命名方式所示：

```
num11=123  
num1="123"
```

在上面的实例中，第一个是使用num1再加上数字1组成，而第二个只有字符num1来作为变量名。这两种方式在书写形式上非常相似，在同时使用时，极易造成变量名称的混淆，从而使得脚本运行错误。

(2) 在命名变量时，如果使用一个单词不能表达变量的用途，可以使用多个单词联合表示，单词之间可以使用下划线连接，也可以将每个单词的首字母大写。但是变量名的长度不能太长，一般不要超过80个字符。因为一旦使用了太多字符，将会降低程序的可读性。如下面的命名方式所示：

```
temp_num=123  
TempNum=456
```

第一行使用下划线连接的形式来表示变量temp_num，而第二行使用首字母大写而不添加下划线的方式来表示变量。这两种方式没有任何区别，都能正确地表示变量的作用，并且能够通过变量的名称来大体上判断变量的作用以及使用范围。

(3) 命名时要尽量做到函数名称和实际功能相符合。尽量不要使用fun1()、fun2()这种名称极为接近的形式来进行命令，否则，一旦使用的函数过多，所有的函数名又极其相似，在进行函数的调用时，会使得脚本的可读性非常差。

注意

在本书中只是为了对知识点进行必要的说明，而没有任何的意义。所以经常会出现使用fun1()、fun2()作为函数名的情形。但是希望读者不要使用此种方式。

10.1.2 注释风格

在Shell脚本中，只支持一种注释方式，就是使用“#”来对井号后面的内容进行注释。从“#”所在的位置开始，到该行的末尾，中间所有的内容都被当作是注释的内容。在执行Shell脚本时，注释中的内容是不会被执行的。关于井号在Shell脚本中的使用如示例脚本10-1.sh所示。

```
#示例脚本10-1.sh    井号的使用
#!/bin/bash
```

```
echo 在注释中使用单引号
echo this # is 'is'
```

```
echo 在注释中使用双引号
echo this # is "is"
```

```
echo 在注释中使用倒引号
echo this # is
```

```
echo 输出一个井号
echo this \#
```

为示例脚本10-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-1.sh
ben@ben-laptop:~$ ./10-1.sh
在注释中使用单引号
this
在注释中使用双引号
this
在注释中使用倒引号
this

输出一个井号
this \#
```

通过示例脚本10-1.sh的运行结果可以看出，在使用了井号之后，不管其后面使用何种符号，都被Bash作为注释的内容而在执行脚本的时候忽略掉。而如果需要输出一个井号，就需要使用转义字符反斜杠“\”来将井号转义成普通的字符，从而输出一个井号。如果不使用转义字符，井号将被视为注释的开始。

在Shell脚本中，如果存在连续的多行语句需要注释，可以选择在每一行的开头都添加井号，如果需要注释的范围比较大，工作量也是比较大的。在此时，可以使用函数的形式间接地注释，如示例脚本10-2.sh所示。

```
#示例脚本10-2.sh    使用函数进行注释
```

```
#!/bin/bash

echo 脚本开始
echo 使用函数进行注释
fun()
{
echo this # is 'is'

echo 在注释中使用双引号
echo this # is "is"

echo 在注释中使用倒引号
echo this # `is` 

echo 输出一个井号
echo this \#
}

echo 不执行函数
echo 脚本结束
```

为示例脚本10-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-2.sh
ben@ben-laptop:~$ ./10-2.sh
脚本开始
使用函数进行注释
```

不执行函数

脚本结束

通过示例脚本10-2.sh的运行结果可以看出，如果出现大段的连续语句需要注释，可以将这些内容放到一个函数中，而在脚本中不去执行这个函数，就得到了和使用井号同样的效果。

10.1.3 其他需要注意的规范

在编写脚本时，要在适当的地方使用空格和空行。正确地使用空格和空行可以提高脚本的可读性，而在输出结果中使用空行，可以使得输出结果一目了然。

在脚本中出现的模块一般会使用函数或者由功能相对独立的命令组合来完成。而在这些单独的模块与模块之间一般使用一个空行来将不同的模块进行分隔，从而使整个脚本的结构比较清晰。关于空行的使用如示例脚本10-3.sh所示。

```
#示例脚本10-3.sh    脚本中使用空行
#!/bin/bash

fun1()
{
    echo 函数fun1
}
```

```
echo 调用函数fun1
```

```
fun1
```

```
echo 执行其他语句
```

```
echo hello world
```

为示例脚本10-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-3.sh
```

```
ben@ben-laptop:~$ ./10-3.sh
```

```
调用函数fun1
```

```
函数fun1
```

```
执行其他语句
```

```
hello world
```

在上面的脚本中，在说明使用的Shell类型和其他的内容之间使用空行来使Shell类型比较醒目，使读者能够第一眼看到使用的Shell类型。在函数fun1()的实现部分前后都使用了空行，从而使得当用户在查找函数时能够非常方便地找到。而在调用函数和其他的语句之间使用一个空行，将函数和其他的语句进行分隔，从而使执行函数的地方清晰可见。

空格可以在脚本中突出某些特定的内容，从而使其他人一看就会注意到这部分内容，如某些关键字以及关键命令。但是在某些场合就不需要使用空格，如在变量赋值时不要在赋值运算符等号两边使用空格。而在进行逻辑判断时，测试命令“[]”的前后都需要使用空格。

注意

使用空格和空行需要遵守Shell脚本编写语法，这是所有规范的前提。

10.2 脚本优化

Shell脚本依赖于Shell命令，并且对脚本中出现的命令顺序执行，而非并行执行，这在一定程度上对系统资源是一种浪费。而如果脚本编写得不合适，那么会使系统资源造成巨大的浪费。因此，在编写Shell脚本时，要有意识地对脚本进行优化，防止不必要的资源浪费。

注意

脚本既定功能实现并不代表问题的处理结束，而是代表对脚本进行优化的开始，从而使得问题的处理更加完美。

10.2.1 Shell脚本优化原则

在编写Shell脚本时，使用内置命令，尽量不要使用外置命令。所谓内置命令就是内置Linux系统源码中的命令，Bash在执行这部分命令时，可以不用去磁盘中查找而直接执行，因此其执行速度要快于外部命令。而使用外部命令时，会首先创建新的进程，然后在新的进程中执行命令，这样就会使CPU和内存资源造成浪费。

注意

在Bash中，可以使用**type**命令查看命令是否为内置命令。

在必要的情况下，尽量减少“|”（管道）的使用，因为管道是很耗资源的。当数据量较多或执行次数较多时，资源消耗会非常明显，如示例脚本10-4.sh所示。

```
#示例脚本10-4.sh    使用管道实例
#!/bin/bash

fun1()
{
    for ((i=0;i<10000;i++))
    do
        echo hello world | tee -a data.txt
    done
}

fun2()
{
    for ((i=0;i<10000;i++))
    do
        echo hello world >> data.txt
    done
}
```

```
echo 记录函数fun1执行时间
```

```
time fun1
```

```
echo
```

```
echo 记录函数fun2执行时间
```

```
time fun2
```

为示例脚本10-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-4.sh
```

```
ben@ben-laptop:~$ ./10-4.sh
```

```
记录函数fun1执行时间
```

```
real    0m30.565s
```

```
user    0m18.858s
```

```
sys     0m10.831s
```

```
记录函数fun2执行时间
```

```
real    0m0.399s
```

```
user    0m0.175s
```

```
sys     0m0.077s
```

通过示例脚本10-4.sh的执行结果可以看出，在执行了10000的操作时，使用管道符比使用重定向符所使用的时间要差几百倍。因此如果在脚本中存在大量类似的重复操作时，要尽量避免使用管道，而使用其他的类似操作来完成，避免使CPU和内存资源造成不必要的浪费。

在进行字符串的操作时，尽量不要使用awk、sed等高级编辑工具，因为这些编辑工具也需要创建新的进程从而完成字符串的操作。

10.2.2 提供足够的提示信息

Shell脚本是一种交互性很强的脚本语言，在脚本的执行过程中需要进行必要的信息提示，如提示用户输入数据、在展示输出结果时进行必要的说明。如果没有足够的提示信息说明，那么在脚本执行过程中，不管是需要用户输入数据，还是输出信息，都会显得难以理解，如示例脚本10-5.sh所示。

```
#示例脚本10-5.sh    不添加提示性信息

#!/bin/bash
read num
echo num

read num2
echo num2
echo $(( $num+$num2 ))
```

为示例脚本10-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-5.sh
ben@ben-laptop:~$ ./10-5.sh
```

```
10  
10  
10  
10  
20
```

通过示例脚本10-5.sh的运行结果可以看出，如果不知道脚本需要进行何种操作，在脚本执行时用户将不知道进行的操作是什么意思，脚本的执行也变得没有意义。因此在脚本中需要添加一些提示性的信息，如提示用户当前需要进行何种操作，并将展示出来的结果进行详细的说明，如示例脚本10-6.sh所示。

```
#示例脚本10-6.sh    添加提示性信息  
  
#!/bin/bash  
  
echo -n 输入一个整数:  
read num  
echo 输入的整数是: $num  
  
echo -n 输入一个整数:  
read num2  
echo 输入的整数是: $num2  
echo 两个数相加的和为$(( $num+$num2 ))
```

为示例脚本10-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-6.sh
ben@ben-laptop:~$ ./10-6.sh
输入一个整数: 10
输入的整数是: 10
输入一个整数: 10
输入的整数是: 10
两个数相加的和为20
```

通过示例脚本10-6.sh的运行结果可以看出，在添加了适当的提示性信息以后，脚本在执行时出现的每一个数据的作用显得非常明显，相对于示例脚本10-5.sh来说，其可读性和易懂性要高很多。

注意

提示性信息不是越多越好，而是在关键或重要的地方添加适当的提示性信息即可。

10.3 脚本使用技巧

在使用Shell脚本进行操作时，存在一些常用的技巧和方式，可方便脚本的编写。本小节将重点介绍一些通用的命令选项以及如何在Shell脚本中进行算术运算，对于其他方面的内容，读者可以参照其他书籍进行学习。

10.3.1 命令选项说明

在Linux系统中，脚本的编写依赖于Shell命令。而对于众多Shell命令来说，每一个命令都有很多的选项，看似很多的选项之间存在着千丝万缕的联系，某些常用的选项，对于不同的命令来说，其作用是相同的，不同的是展示出来的结果依赖于具体执行的命令。这些常用的选项及其作用如表10-1所示。

表10-1 Shell命令的常用选项

选项	作用	选项	作用
-a	显示全部内容	-c	执行计数功能
-d	指定目录	-e	将内容展开
-f	从指定的文件中获取文件	-h	获取帮助信息
-r	以递归的方式处理目录文件	-y	设置所有的问题的回答为yes
-v	获取命令的版本信息	-i	忽略大小写

表10-1中展示了一些常用的选项的作用，对于大部分命令来说，可以按照这些常用的选项来使用这些命令，对于某些命令的特殊选项用法，可以使用man命令先查看如何使用，在明确了命令的使用方式以后，再使用命令完成既定的操作。

10.3.2 算术运算

在Shell脚本中，所有变量的默认存储方式都是字符串，而其操作方式也是按照字符串进行处理的，而在实际使用过程中，不可避免地需要进行数值的各种算术操作，如数值的加减等，如果仍然按字符串的操作来进行算术操作，那么得到的结果往往不是期望的结果，如示例脚本10-7.sh所示。

```
#示例脚本10-7.sh 算术运算的默认处理方式
#!/bin/bash

echo 使用默认存储方式进行算术运算
echo 输入加数:
read num1

echo 输入另外一个加数:
read num2

sum=$num1+$num2
echo 算术运算的结果为$sum
```

为示例脚本10-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-7.sh
ben@ben-laptop:~$ ./10-7.sh
使用默认存储方式进行算术运算
```

```
输入加数:
```

```
10
```

```
输入另外一个加数:
```

```
20
```

```
算术运算的结果为10+20
```

通过运行结果可看出，在Shell脚本中，变量的默认操作方式是字符串，因此本意是计算两个数相加的和，而最终的结果却将整个操作变成一个字符串。在Shell脚本中，要实现数值的算术运算可以使用如下方式实现：

- **let**命令
- **(())**命令
- **expr**表达式

let命令常用于整数的算术运算，还可以使用长的整数运算，支持除`++`、`--`和逗号`(,)`之外的所有整型运算，**let**命令的一般使用方式如下：

```
let 算术表达式
```

在**let**命令的一般形式中，算术表达式的使用方式和C语言中算术表达式的使用方式类似，可以不必使用\$来标识变量，**bash**会自动进行变量数值的计算，并且计算出最终的结果，**let**命令的使用方式如示例脚本**10-8.sh**所示。

```
#示例脚本10-8.sh let命令的使用
```

```
#!/bin/bash

echo 使用命令let进行算术运算
echo 输入加数:
read num1

echo 输入另外一个加数:
read num2

let sum=num1+num2
echo 算术运算的结果为$sum
```

为示例脚本10-8.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-8.sh
ben@ben-laptop:~$ ./10-8.sh
使用命令let进行算术运算
输入加数:
10
输入另外一个加数:
20
算术运算的结果为30
```

通过实例脚本10-8.sh的运行结果可以看出，在使用了let命令之后，得到的结果不再是字符串的拼接，而是真正的算术运算。

注意

使用let命令时，表达式必须是完整的，并且算术运算符两边不允许出现空格或其他的字符。

在Shell脚本中，可以使用命令“\$(())”来代替let命令，其基本使用方式如下：

((算术表达式))

在Bash中，命令\$(())和let命令相同。在使用时，算术表达式可以只有表达式的右边部分，即表达式的右值。该命令的使用方式如示例脚本10-9.sh所示。

```
#示例脚本10-9.sh 命令(( ))的使用
#!/bin/bash

echo 使用符号(( ))进行算术运算
echo 输入加数:
read num1

echo 输入另外一个加数:
read num2

echo 算术运算的结果为$((num1+num2))
```

为示例脚本10-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-9.sh
ben@ben-laptop:~$ ./10-8.sh
使用命令(( ))进行算术运算
输入加数:
10
输入另外一个加数:
20
算术运算的结果为30
```

通过示例脚本10-9.sh的运算结果可以看出，使用了符号“(())”之后，可以不必使用完整的算术运算表达式来实现数值的算术运算，从而简化了算术运算表达式。

注意

符号“(())”和命令let在运算级别和运算速度上是相同的，而且可以进行替换。但是要注意书写方式的不同。

10.3.3 高级算术运算

在上一小节中介绍了Bash中整数的算术运算，而对于其他类型的数值的操作需要使用bc来完成。bc是Bash中存在的一个高级数学运算工具，不但可以实现浮点数的运算，还可以借助其中的函数实现更加复杂

的操作。**bc**操作的基本形式如下：

```
算术表达式|bc
```

使用**bc**工具时一般是借助管道符来实现各种算术运算的，算术表达式一般使用一对双引号括起来，从而和一般的表达式进行区分。最后面的**bc**不必使用双引号括起来，当整个命令执行完毕后，整个算术运算就完成了，运算结果将输出到屏幕上。**bc**工具的基本使用方式如下：

```
ben@ben-laptop:~$echo "1.20*3.40"|bc  
4.08
```

在上面的表达式中，使用**bc**工具实现了最基本浮点数的运算，可以使用**bc**，还可以使用一些特定的函数来实现更高级的处理，如计算一个数平方根等，计算平方根可以使用函数**sqrt**，如下所示：

```
ben@ben-laptop:~$echo "sqrt(10000)"|bc  
100
```

对于一个程序员来说，数值转换是经常要处理的问题。使用**bc**工具同样可以实现各种数制的转换。在**bc**中，可使用参数**obase**来完成数制的设定，如下所示：

```
ben@ben-laptop:~$echo "obase=2;10"|bc  
1010
```

关于bc的使用实例如示例脚本10-10.sh所示。

```
#示例脚本10-10.sh bc工具的使用
#!/bin/bash

echo -n 输入一个浮点数:
read num

echo 计算浮点数的乘方运算
echo "$num^2" |bc

echo 改为二进制显示
echo "obase=2;$num" |bc
```

为示例脚本10-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 10-10.sh
ben@ben-laptop:~$ ./10-10.sh
输入一个浮点数: 1.2
计算浮点数的乘方运算
1.4
改为二进制显示
1.001
```

通过示例脚本10-9.sh的执行结果可以看出，使用bc工具能够实现浮点数的各种复杂算术运算，除了示例脚本中的操作之外，bc工具还可以

进行其他的操作。

10.4 小结

本章主要介绍了编写Shell脚本时的一些注意事项。Shell脚本作为一门编程语言，同样需要遵守各种规范，同时在编写时，也存在一些技巧。

在编写脚本时，为了提高脚本的可读性，在进行变量、函数等的命名时，要注意使用一些通用的名称，可以按照一般编程语言中使用的匈牙利命令法来进行命名，而不要随意指定变量、函数等的名称。在编写脚本的过程中，要使用足量的注释来对脚本的执行过程或逻辑结构进行必要的说明。

Shell脚本中的命令都是按照顺序执行的，因此为了节省内存资源和CPU资源，尽量使用不会创建新进程而直接执行的方式来完成既定的操作，尽量减少管道、awk、sed等方式的使用。

Shell命令在使用时需要和某些选项一起使用，从而获取到命令执行的特殊结果。在Linux系统中，很多命令的选项作用是类似的，因此可以根据已知命令的选项来判断不熟悉的命令的选项的使用方式。

在Shell中，算术运算是一类特殊的运算方式，因为在Shell中，所有的操作都被认为是对于字符串的操作。要进行整数的算术操作可以借助于let命令、“()”命令和expr表达式，还可以使用bc工具来实现对浮点数的操作。

第11章 正则表达式

在前面的章节中，曾经简单地介绍过如何使用星号“*”来匹配所有的内容，如对于数组来说可以表示数组中所有的元素。这种使用一个符号来表示，而可以匹配一系列符合相应规则的字符串的表达式，一般被称为正则表达式。在使用正确的前提下，可以方便地检索、替换那些符合某个模式的字符串或文本。而在Shell脚本中，正则表达式也有着非常广泛的应用。

本章的主要内容如下：

- 正则表达式的基本介绍
- 正则表达式中的常用符号
- 正则表达式的实战练习

11.1 正则表达式基础

正则表达式现在广泛应用于Unix系统、Linux系统以及PHP、C#、Java等开发环境中，在其他的编程语言中也会用到正则表达式。因为使用正则表达式可以通过简单的方法来实现比较复杂的功能。因此，好好地学习正则表达式可以极大简化日常的工作。但是学习的过程将是非常艰辛的，需要广大读者付出一定的努力。

11.1.1 正则表达式的定义

正则表达式（Regular Expression）就是使用简单的字符按照预先设定的规则来完成复杂的功能。从根本上来讲，正则表达式是一种字符串的匹配方式，比如使用星号“*”来匹配任意的字符等。使用正则表达式可以从一个字符串中查找某些特殊的字符或字符串，还可以将检索出来的字符串使用其他的字符或字符串进行替换。

正则表达式一般是由普通字符以及特殊字符组成的字符串。普通字符就是常用的大小写字符（a~z和A~Z）、数字、标点符号以及其他可打印字符和非可打印字符。特殊字符就是前面章节中介绍的元字符。正则表达式的组成可以是单个的字符、字符集合、字符范围、字符间的选择或者是这些字符之间的任意组合。

正则表达式实际上是一个匹配的模板，当bash执行该正则表达式时，会将所有的输入数据与匹配模板进行比较，如果符合匹配模板，那么就匹配成功。而那些匹配不成功的数据则被过滤掉，如图11-1所示。

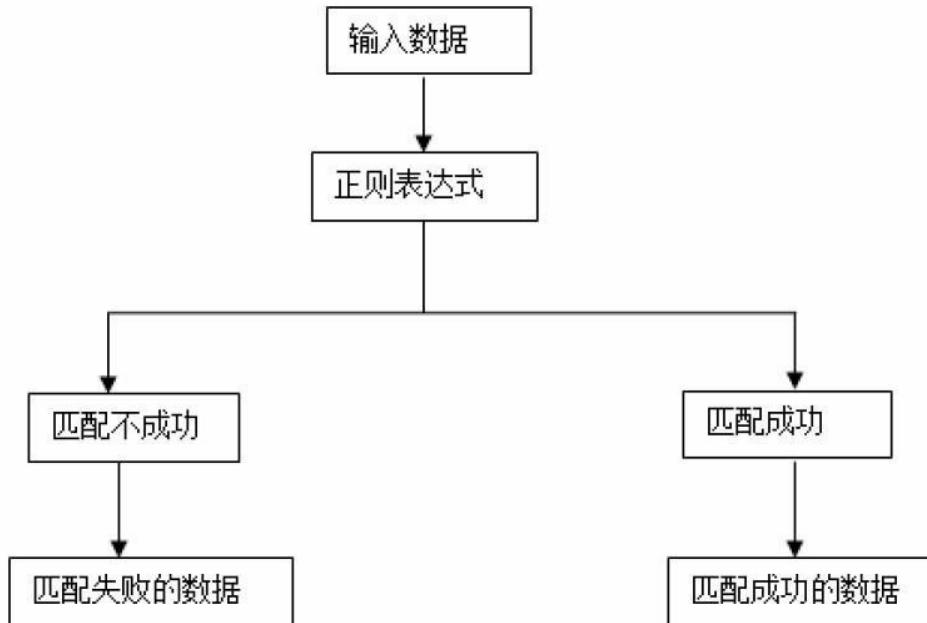


图11-1 正则表达式数据流处理过程

通过图11-1可以看出，正则表达式可以将输入数据按照是否匹配成功模板分成匹配成功和匹配失败两部分。一般来说，只有匹配成功的部分才会显示出来（或者存储到其他介质中）。

11.1.2 正则表达式的分类

在Linux系统中，不同的编程语言（编程环境）需要使用不同类型的正则表达式。而正则表达式的类型由正则表达式引擎控制。正则表达式引擎控制负责解释正则表达式的匹配规则以及使用方式，其作用类似于Shell对于Linux系统。正则表达式引擎一般可以分为以下两种形式：

- POSIX基本正则表达式引擎；
- POSIX扩展正则表达式引擎。

基本正则表达式引擎可以用于Linux系统中的大部分应用程序。这些应用程序都可以识别，并且能够按照符号的既定含义进行匹配。而有些程序因为执行速度的原因，只能支持部分正则表达式。由于不同的程序中使用的正则表达式的内容不一致，所以在本书中，只使用一些常用的正则表达式符号以及使用方式，对那些不常用的内容，将不会涉及，如果读者有兴趣进行深层次的学习，可以参考相关的资料。

11.2 基本正则表达式的常用符号

基本的正则表达式就是匹配输入数据流中的字符信息，在进行字符的匹配时，一般使用一些特殊的字符匹配字符串中的某些内容，熟练和正确地使用这些字符可以使编写出来的正则表达式既简单又高效。

11.2.1 使用点字符匹配单字符

点字符（.）是英文中的点字符，对应于中文中的句号。该符号可以匹配任意单个字符，但是必须存在一个字符。如果当前位置没有字符，那么匹配结果是失败。关于点字符的使用如示例脚本11-1.sh所示。

```
#示例脚本11-1.sh 点字符匹配单字符
#!/bin/bash

#Hello
#HEllo
#hEllo
echo 使用点字符匹配字符
grep -v H.llo 11-1.sh
echo

grep ..llo 11-1.sh
```

为示例脚本11-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-1.sh
ben@ben-laptop:~$ ./11-1.sh
使用点字符匹配字符
#Hello
#HEllo
#hEllo
```

```
grep .llo 11-1.sh
```

通过示例脚本11-1.sh的执行结果可以看出，使用点符号能够匹配一个任意的字符。而且使用点号可以将不关心的字符排除掉，只出现比较重要的字符。

注意

正则表达式需要和其他的检索命令一同使用，本章使用grep命令来作为示例进行讲解。

11.2.2 使用定位符

在日常生活中，我们会遇到判断某些数据开头字符的情况，如判断数字串是不是以123开始的，而对于开头字符（或字符串）以外的数据内容，不需要关心。此时就需要使用脱字符（^）来进行匹配。使用脱字符的一般方式如下：

```
^string
```

对于上面的表达式，如果字符串的开头字符为string，那么匹配成功。而即使在字符串的其他位置找到匹配字符串string，匹配也是失败。关于脱字符的使用实例如示例脚本11-2.sh所示。

```
#示例脚本11-2.sh 点字符匹配单字符
```

```
#!/bin/bash

#Hello
#123HEllo #123hEllo
echo 显示开头是#1的
grep [^1] 11-2.sh
echo
echo 显示开头是H的
```

为示例脚本11-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-2.sh
ben@ben-laptop:~$ ./11-2.sh
显示开头是#1的
#123HEllo
#123hEllo
```

在正则表达式中，脱字符“^”和美元符号“\$”一般用来匹配行首字符和行尾字符。在特殊的情况下，如需要匹配开头为脱字符的字符串，在使用时需要使用转义字符，将脱字符转移成普通的字符，然后再使用脱字符进行匹配，从而实现查找开头字符为脱字符的字符串，如示例脚本11-3.sh所示。

```
#示例脚本11-3.sh 查找开头字符为脱字符
#!/bin/bash
```

```
#^  
echo 匹配开头字符为脱字符  
grep "#^" 11-3.sh  
echo  
  
echo 查找包含脱字符的文本行  
grep "*#\^*" 11-3.sh
```

为示例脚本11-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-3.sh  
ben@ben-laptop:~$ ./11-3.sh  
匹配开头字符为脱字符  
#^  
  
查找包含脱字符的文本行  
#^  
grep "#^" 11-3.sh  
grep "*#\^*" 11-3.sh
```

通过示例脚本11-3.sh可以看出，如果要将脱字符视为普通字符，那么需要使用转义字符将其转换成普通字符，然后就可以当做特殊字符进行使用。

注意

脱字符仅表示开头的字符和结尾字符需要匹配哪些字符串，不能

代替其他字符的存在。

除了使用脱字符匹配行首的字符串以外，还可以使用美元符号（\$）来匹配行尾的字符串。该符号的基本使用方式如下：

string\$

上面的表达式是在文本表达式的后面加上美元符号，表示字符串必须以文本string为结尾，即使在文本中的其他的位置存在字符串string，那么匹配的结果也是失败，如示例脚本11-4.sh所示。

```
#示例脚本11-4.sh 使用美元符号匹配行尾字符
#!/bin/bash

#123
#hao123
#hao

echo 使用美元符号匹配结尾字符
grep "[1-9\$]" 11-4.sh
```

为示例脚本11-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-4.sh
ben@ben-laptop:~$ ./11-4.sh
#123
#hao123
```

在实际应用中，一般联合使用脱字符和美元符号来进行匹配。也就是说既匹配行首的字符，也匹配行尾的字符，而不关心中间的字符，如示例脚本11-5.sh所示。

```
#示例脚本11-5.sh 匹配空行
#!/bin/bash

echo 使用定位符匹配空行

grep '^$' 11-5.sh
```

为示例脚本11-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-5.sh
ben@ben-laptop:~$ ./11-5.sh
使用定位符匹配空行

ben@ben-laptop:~$
```

通过示例脚本11-5.sh的执行结果可以看出，使用脱字符和美元符号能够对空行进行匹配，而直接判断文本中是否存在空行。

注意

脱字符和美元符号一般被称为定位符。

11.2.3 使用“*”匹配字符串中的单字符或其重复序列

在进行字符串的匹配时，经常会遇到需要匹配某个字符或字符串的情形，如判断字符串中是否包含子串“ab”和“cd”，而在子串之间还会有个数不同并且内容也不同的字符或字符串，此时需要使用“*”进行匹配。星号一般有3种使用方式：

```
a*  
*a  
a*b
```

星号在字符或字符串的左边和右边的作用一样，表示字符或字符串在文本中出现零次或多次。而对于第3种方式来说，表示字符a和b同时出现，而在a和b之间可以存在零个字符，也可以存在多个字符。星号的使用使得正则表达式的表示范围大大增加。关于星号的使用方式，如示例脚本11-6.sh所示。

```
#示例脚本11-6.sh 使用*匹配字符  
#! /bin/bash  
  
echo 显示包含11的所有文件  
ls -l *11*  
echo
```

```
echo 显示以11开头的文件
```

```
ls -l 11*
```

```
echo
```

```
echo显示以11结尾的文件
```

```
ls -l *11
```

为示例脚本11-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-6.sh
```

```
ben@ben-laptop:~$ ./ 11-6.sh
```

```
显示包含11的所有内容
```

```
11-1.sh 11-2.sh 11-2.sh 11-4.sh 11-5.sh 11-6.sh
```

```
显示以11开头的文件
```

```
11-1.sh 11-2.sh 11-2.sh 11-4.sh 11-5.sh 11-6.sh
```

```
显示以11结尾的文件
```

通过示例脚本11-6.sh的执行结果可以看出，使用“*”可以匹配任意多个字符，可以使用该符号来忽略那些无关紧要的所有的字符，而只匹配比较重要的字符信息，从而使得进行匹配时，重点和非重点比较明显。

11.2.4 使用“\”屏蔽一个特殊字符

的含义

在正则表达式中，特殊字符都表示特殊的含义，如脱字符表示从行首开始匹配，而美元符号表示从行尾开始匹配。而如果需要匹配这些特殊字符，就要使用转义字符“\”将特殊字符进行转义，使其变成一个普通的字符。关于转义字符的使用方式如示例脚本11-7.sh所示。

```
#示例脚本11-7.sh 屏蔽特殊字符
#!/bin/bash

grep "*\$*" 11-7.sh
echo

echo '不使用反斜杠匹配$'
grep "*$*" 11-7.sh
```

为示例脚本11-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-7.sh
ben@ben-laptop:~$ ./ 11-7.sh
使用反斜杠匹配$
grep "*\$*" 11-7.sh

不使用反斜杠匹配$
grep "*$*" 11-7.sh
```

通过示例脚本11-7.sh的运行结果可以看出，如果要匹配特殊字符，需要使用转义字符来将特殊字符转义成普通字符，然后就可以进行任意使用了。

11.3 扩展正则表达式的常用符号

对于正则表达式来说，扩展的正则表达式也常用于实际的操作中，常用的扩展正则表达式符号为“[]”。本小节将重点介绍该符号的使用方式。

11.3.1 使用“[]”匹配一个范围或集合

在使用正则表达式时，经常会遇到在一定范围内进行字符或字符串匹配的情况，如判断某个字符是不是数字（数字的范围从0到9），或是某个字符是不是字符（大写字符或是小写字符），如果需要匹配的内容较少，可以将所有的字符都写出来。而当需要匹配的字符个数过多时，则需要使用一个新的字符，即一对中括号“[]”，在中括号中需要添加匹配字符的集合，如下列表达式所示：

```
[a-c]
```

```
[5-9]
```

在上面的两个表达式中，第1个表达式表示需要匹配的字符为a、

b、c，对于其他的字符都属于匹配失败。第2个表达式表示需要匹配的内容是数字5、6、7、8、9，对于其他字符的匹配也属于匹配失败。关于范围匹配符号的使用如示例脚本11-8.sh所示。

```
#示例脚本11-8.sh 使用[]匹配一个范围
#!/bin/bash

echo 匹配开头字符是字符
grep "[a-z]^" 11-8.sh
```

为示例脚本11-8.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-8.sh
ben@ben-laptop:~$ ./ 11-8.sh
匹配开头字符是字符
echo 匹配开头字符是字符
grep [a-z] 11-8.sh
```

通过示例脚本11-8.sh的运行结果可以看出，如果需要匹配的字符是连续的字符，如都是字母或数字，那么可以使用一对中括号（[]）来表示某一段范围，从而使正则表达式更加简单明了。

11.3.2 使用“{}”匹配模式结果出现的次数

在使用星号或其他符号时，只要输入数据中至少有一次出现正则表达式中的模板，就算是匹配成功。而如果需要查找匹配模板出现3次或多次，就需要使用符号“\{\}”来判断匹配模板出现的次数。该符号的使用方式如下：

```
string \{ n \}
string \{ n, \} m
string \{ n, m \}
```

在上面3个表达式中，第1个表达式用来匹配string出现的次数，如果出现n次，那么匹配成功。对于第2种方式来说，和第1种含义相同，但是string出现的次数最少为n次。对于第3种方式来说，string出现次数必须在n与m之间，小于n次或是大于m次都不属于匹配成功，如示例脚本11-9.sh所示。

```
#示例脚本11-9.sh 模式出现几率的使用
#!/bin/bash

echo 匹配字符b至少出现2次
grep "b\{2\}" 11-9.sh
echo

echo 匹配字符c出现1~2次
grep "c\{1,2\}" 11-9.sh
```

为示例脚本11-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-9.sh
ben@ben-laptop:~$ ./ 11-9.sh
匹配字符b至少出现2次
#!/bin/bash

匹配字符c出现1~2次
echo 匹配字符b至少出现2次
echo
echo 匹配字符c出现1~2次
grep "c\{1,2\}" 11-9.sh
```

通过示例脚本11-9.sh的运行结果可以看出，在正则表达式中可以对某些字符出现的次数进行精确计数，从而能够对文本中的数据进行精准定位。

11.3.3 问号的使用

在进行重复匹配时，经常使用星号来匹配模板中的字符或字符串出现一次或是多次的情形。除此之外，还使用问号来匹配那些只出现一次或没有出现的情形。当使用问号时，匹配模板出现一次或是没有出现，则匹配成功。而如果匹配模板出现的次数超过一次，那么匹配失败。问号的使用方式如下：

```
ab?c
```

```
12?3
```

在上面的第一个表达式中，如果输入数据中存在ac、abc之类的字符串，那么该数据匹配成功，因为字符b出现了零次或一次。而如果abbcc、abbc之类的字符串中出现超过一次的字符，那么此时匹配是失败的。关于问号的使用实例如示例脚本11-10.sh所示。

```
#示例脚本11-10.sh 问号的使用
#!/bin/bash

echo 匹配出现过字符b的文本
grep "*b?" 11-10.sh

echo
echo 匹配出现过字符e的文本
grep "*e?" 11-10.sh
```

为示例脚本11-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 11-10.sh
ben@ben-laptop:~$ ./ 11-10.sh
匹配出现过字符b的文本
grep "*b?" 11-10.sh

匹配出现过字符e的文本
e
grep "*e?" 11-10.sh
```

通过示例脚本11-10.sh的运行结果可以看出，使用问号能够保证文

本数据至少出现一次，而不像星号那样，即使不出现匹配的文本，也会记为匹配成功。

注意

在扩展正则表达式中，还经常使用加号来表示匹配模板出现至少一次的情形。其使用方式和星号、问号类似。

11.4 小结

本章主要介绍了正则表达式在Shell脚本中的使用，重点介绍了基本正则表达式以及扩展正则表达式中常用符号的使用方式。正则表达式一般用来处理文字数据，可以从一个字符串中查找某些特殊的字符或字符串，还可以将检索出来的字符串用其他的字符或字符串进行替换。

正则表达式可以分为基本正则表达式和扩展正则表达式。在基本正则表达式中，一般使用点字符来匹配任意的一个字符，而使用星号来匹配零个或多个字符，脱字符和美元符号作为定位符，常用来确定字符串出现在行首或行尾。而如果需要将特殊字符作为普通字符进行匹配，需要使用转义字符反斜杠来进行转义。

对于扩展正则表达式来说，中括号可以用来表示字符串出现的范围，一般是从低到高进行表示，而使用问号来表示匹配模板出现一次或多次，而加号常用来表示出现至少一次。

使用特殊字符和普通字符的结合，可以通过定义相应的正则表达式

来作为相应的匹配模板，然后可以使用**sed**编辑器或**gawk**编辑器来从输入数据流中进行筛选，从而获取符合要求的数据。

第12章 Shell中的文本搜索工具 ——grep家族

在上一章中介绍的正则表达式是对文本数据按照模板进行匹配，在进行文本匹配时，首先要进行文本的搜索，在Linux系统中，常用的文本搜索工具是grep命令，本章将重点介绍grep命令的使用方式。

本章的主要内容如下：

- grep的基本使用方式以及常用选项的使用
- grep命令和正则表达式的协同使用
- grep命令命令和系统命令的协同使用

12.1 grep的基础使用

在Linux系统中，经常需要进行文本的查找，如在目录中查找某个文件，或者是查找文件中的特定内容。而经常使用的命令是grep命令，该命令可以作为文本搜索工具来实现文本的检索。

12.1.1 grep命令的基本使用方式

grep命令作为Linux系统中常用的文本检索工具，有着非常广泛的用途。其一般形式如下：

```
grep [选项] [匹配样本] [文件列表]
```

grep 命令一般用于在指定的文件列表中查找指定的匹配样本。如果指定文件中的内容包含符合所指定的样式，那么含有样式的那一列将会显示出来（一般显示到屏幕上，也可使用重定向显示到其他的地方）。如果没有找到指定的内容，那么将不显示任何内容。在使用grep命令时，如果不指定文件名列表，或使用的文件名为“-”，那么grep指令会从标准输入设备读取数据进行匹配。grep命令的简单使用实例如示例脚本12-1.sh所示。

```
#示例脚本12-1.sh  grep命令的使用
#!/bin/bash

echo "grep命令的简单使用示例"
echo "在当前目录中查找文件"
grep "12-1.sh"./
echo 找到文件12-1.sh
echo "使用grep命令查找不存在的文件"
grep "no_exsit.txt" ./
echo 未找到不存在的文件
```

为示例脚本12-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-1.sh
ben@ben-laptop:~$ ./ 12-1.sh
grep命令的简单使用示例
```

在当前目录中查找文件

找到文件**12-1.sh**

使用**grep**命令查找不存在的文件

未找到不存在的文件

通过示例脚本12-1.sh的执行结果可以看出，当使用**grep**命令时会出现多个结果，并且使用一次**grep**命令不能检索出需要的结果时，可以将**grep**命令和管道符一起使用，从而使上一次检索的结果能够继续被检索，最终得到需要的结果。关于**grep**命令和管道符的使用方式如示例脚本12-2.sh所示。

```
#示例脚本12-2.sh  grep命令和管道符的使用
#!/bin/bash

echo "多次使用grep命令，逐次缩小范围"
ls -l | grep 12-1.sh
```

为示例脚本12-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-2.sh
ben@ben-laptop:~$ ./ 12-2.sh
多次使用grep命令，逐次缩小范围
-rwxrwxrwx 1 ben ben 177 2014-03-18 18:40 12-1.sh
```

从示例脚本12-2.sh的执行结果中可以看出，在使用**grep**命令之前首先执行**ls -l**命令，而**grep**命令在**ls**命令的执行结果中进行检索，当检索到

有相应的记录时就进行显示，而如果没有找到合适的内容，则不会进行任何显示。

注意

在使用grep命令时，一定要提供一个文件的检索路径，否则grep命令会一直等待用户输入，直至程序中断。

12.1.2 grep选项

grep命令经常和一些选项一起使用，从而使对内容的查找更加灵活。grep命令常用的选项如表12-1所示。

表12-1 grep命令常用选项

选项名称	作用
-c	只输出匹配行的计数，不显示匹配的内容
-i	不区分大小写（只适用于单字符）
-h	查询多文件时不显示文件名
-n	显示匹配行及行号
-s	不显示不存在或无匹配文本的错误信息
-v	显示不包含匹配文本的所有行

-E

允许使用扩展模式匹配

从表12-1中可以看出，在使用grep命令时，可以通过-c选项只输出匹配的行数，而不显示具体的内容。使用-l选项可在匹配时对字符的大小写不敏感。使用-h选项可在查询多个文件时不显示文件名。而使用-n选项可以显示匹配的行号以及该行的内容。使用选项-s来不显示不存在或无匹配文本的错误信息。使用选项-v显示不包含匹配文本的所有行。关于这些选项的详细用法，将在后面详细讲述。

注意

表12-1中出现的选项只是grep命令常用的选项，关于grep命令的其他选项，可以使用man grep命令进行查看。

12.1.3 行数

在使用grep命令时，有时候仅需要知道最终匹配的行数，而至于哪些是需要的内容，则不用关心。此时，需要使用grep命令的-c选项，忽略匹配的内容，而只显示行号，关于-c选项的使用方式如示例脚本12-3.sh所示。为了更能直观地观察-c选项的使用方式，在示例脚本中先将匹配的内容显示处理，然后再使用-c选项来显示匹配的行数，从而起到比较的作用。

```
#数据文件test.txt  
hello123
```

```
hello123
Hello123

#示例脚本12-3.sh  grep命令中选项的使用

#!/bin/bash

echo '显示匹配hello的内容'
grep -E "hello" test.txt
echo 显示匹配的行数
grep -c "hello" test.txt
echo

echo '显示匹配Hello的内容'
grep -E "Hello" test.txt
echo 显示匹配的行数
grep -c "Hello" test.txt
```

为示例脚本12-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-3.sh
ben@ben-laptop:~$ ./ 12-3.sh
显示匹配hello的内容
hello123
Hello123
显示匹配的行数
```

```
显示匹配Hello的内容
```

```
Hello123
```

```
显示匹配的行数
```

```
1
```

从示例脚本12-3.sh的执行结果中可以看出，在使用-c选项后，grep命令只显示匹配的行数。而使用了-E选项之后，grep命令对字符的大小写就变得敏感了，字符“h”和“H”就变成了两个字符。

12.1.4 大小写敏感

在进行文本匹配时，一般情况下是使用什么模板就需要匹配什么样的内容，即对文本中字符的大小写是非常敏感的。而在有些时候则不需要关心字符的大小写，只要出现这个字符就是匹配成功，如在文件/dev/passwd中查找“h”或“H”字符时，可以使用下列命令表示：

```
grep 'h' 'H' /dev/passwd
```

上面的表达式是当使用的字符比较少时，可以将需要匹配的字符逐个写出来，而如果需要查找的是“hello”字符串，那么逐个写出所有出现的字符可能是不现实的。这时就需要使用-l选项，使得在进行查找时，不再关心字符串的大小写，以检索出需要的内容，如示例脚本12-4.sh所示。

```
#示例脚本12-4.sh  grep命令中选项的使用
```

```
#! /bin/bash
```

```
#hello
```

```
#HELLO
```

```
echo 不使用-E选项不区分大小写
```

```
echo 匹配hello
```

```
grep *hello* 12-4.sh
```

```
echo
```

```
echo 匹配HELLO
```

```
grep *HELLO* 12-4.sh
```

```
echo
```

```
echo 使用-E选项区分大小写
```

```
echo 区分大小写匹配hello
```

```
grep -E *hello* 12-4.sh
```

为示例脚本12-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-4.sh
```

```
ben@ben-laptop:~$ ./ 12-4.sh
```

```
不使用-E选项不区分大小写
```

```
匹配hello
```

```
grep *hello* 12-4.sh
```

```
grep -E *hello* 12-4.sh
```

匹配HELLO

```
grep *HELLO* 12-4.sh
```

使用-E选项区分大小写

区分大小写匹配hello

```
#hello
```

```
echo 匹配hello
```

```
grep *hello* 12-4.sh
```

```
echo 区分大小写匹配hello
```

```
grep -E *hello* 12-4.sh
```

从示例脚本12-4.sh的执行结果可以看出，如果不使用-E选项，grep命令默认是不关心字符的大小写的，在匹配“hello”和“HELLO”时，其效果是类似的。这样在精确度较高的情况下，就需要对字符的大小写进行区分，而在使用了-l选项之后，“hello”和“HELLO”就变成了两个字符串了。

12.1.5 显示非匹配行

在使用grep命令时，如果在文本中找到匹配的内容，那么就会将相应的内容输出，如果使用了-v选项，那么就会进行反向匹配，原本匹配成功的内容不再显示，而显示出来的是匹配不成功的部分。关于-v选项的使用方式如示例脚本12-5.sh所示。

```
#示例脚本12-5.sh  grep命令中-v选项的使用
```

```
#! /bin/bash
```

```
#hello
```

```
#Hello
```

```
#HELLO
```

```
#123
```

```
#345
```

```
echo 显示匹配行
```

```
grep -E *hello* 12-5.sh
```

```
echo 使用-v选项显示不匹配行
```

```
grep -v *hello* 12-5.sh
```

```
grep -v *HELLO* 12-5.sh
```

```
echo 显示开头字符不是字母的数据
```

```
grep -v ^[a-zA-Z] 12-5.sh
```

为示例脚本12-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-5.sh
```

```
ben@ben-laptop:~$ ./ 12-5.sh
```

```
显示匹配行
```

```
#hello
```

```
grep -E *hello* 12-5.sh
```

```
grep -v *hello* 12-5.sh
```

```
使用-v选项显示不匹配行
```

```
#!/bin/bash

#hello
#Hello
#HELLO
#123
#345
echo 显示匹配行
echo 使用-v选项显示不匹配行
grep -v *HELLO* 12-5.sh
```

```
echo 显示开头字符不是字母的数据
grep -v ^[a-zA-Z] 12-5.sh

#!/bin/bash

#hello
#Hello
#HELLO
#123
#345
echo 显示匹配行
grep -E *hello* 12-5.sh
echo 使用-v选项显示不匹配行
grep -v *hello* 12-5.sh
```

```
echo 显示开头字符不是字母的数据
grep -v ^[a-zA-Z] 12-5.sh
显示开头字符不是字母的数据
#!/bin/bash
```

```
#hello  
#Hello  
#HELLO  
#123  
#345
```

由示例脚本12-5.sh的执行结果可以看出，使用-v选项能够实现grep命令的反向输出，即将匹配不成功的数据输出，而匹配成功的数据将不输出。这种情况可以用于对于某些内容的反向判断，如检查文本中错误的、不符合要求的内容等。

12.1.6 查询多个文件或多个关键字

在使用grep命令时，经常会遇到在多个文件中检索模板内容或存在多个模板需要进行检索的情况。对于在多个文件中进行检索的情形来说，如果被检索的文件在一个目录文件中，那么一般使用通配符星号（“*”）来代替需要检索的所有的文件，而当grep命令在运行时，会自动到目录文件中的所有子文件中进行检索，直至查看完所有的文件。而在检索的过程中，遇到匹配的内容，则直接显示出来。关于通配符星号在grep命令中的使用方式如示例脚本12-6.sh所示。

```
#示例脚本12-6.sh  grep命令中选项的使用  
#!/bin/bash
```

```
echo '使用*进行匹配'  
echo "匹配12-6*"  
ls -l | grep 12-6*
```

为示例脚本12-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-6.sh  
ben@ben-laptop:~$ ./12-6.sh  
使用*进行匹配  
匹配12-6*  
-rwxrwxrwx 1 ben ben 79 2014-03-18 20:23 12-6.sh
```

从示例脚本12-6.sh的执行结果中可以看出，使用匹配符星号可以匹配一个或多个任意的字符，在使用时除了重点字符以外，其余都可以使用星号代替。

如果需要检索多个关键字，即需要匹配多个关键字时，一般使用-E选项实现。在使用-E选项时，grep命令就可以在相同的文件中同时匹配多个关键字，关键字一般使用双引号括起来，并且关键字之间使用竖线进行分割。关于grep检索多个关键字的实例如示例脚本12-7.sh所示。

```
#示例脚本12-7.sh  grep检索多个关键字  
#!/bin/bash  
  
#hello  
#Hello  
#HELLO  
#HEllo
```

```
#HEllo

echo '使用grep命令的-E选项'
echo '匹配hello|Hello'
grep -E "hello|Hello" 12-7.sh
echo
echo '匹配HE.lo|hello'
grep -E "HE.lo|hello" 12-7.sh
```

为示例脚本12-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-7.sh
ben@ben-laptop:~$ ./ 12-7.sh
使用grep命令的-E选项
匹配hello|Hello
#hello
#Hello
echo '匹配hello|Hello'
grep -E "hello|Hello" 12-7.sh
echo '匹配HE.lo|hello'
grep -E "HE.lo|hello" 12-7.sh

匹配HE.lo|hello
#hello
#Hello
#HEllo
```

```
echo '匹配hello|Hello'  
grep -E "hello|Hello" 12-7.sh  
echo '匹配HE.lo|hello'  
grep -E "HE.lo|hello" 12-7.sh
```

从示例脚本12-7.sh的执行结果中可以看出，当存在多个关键字时，可以使用-E选项进行匹配，在多个关键字之间需要使用竖线作为分隔符来区分每一个关键字。

注意

在使用grep命令实现检索多个关键字时，其选项一般使用大写的字符E。

12.2 grep和正则表达式

grep命令常和正则表达式一起使用，以满足各种需求的文本检索。当grep命令和正则表达式一起使用时，正则表达式只是字符串的一种描述形式，而grep命令作为一个支持正则表达式的工具，可完成文本的检索。本节将重点讲述grep命令和正则表达式一起使用的方式。

注意

为了在文本检索过程中，使正则表达式的匹配能够顺利进行，正则表达式需要使用引号括起来。

12.2.1 模式范围以及范围组合

在进行文本的检索时，经常需要在一定的范围内查找。而除了使用-E选项之外，还可以使用表示范围的“[]”在一定范围内进行检索。该符号可以单独使用，也可以多个一起使用，以形成一个范围的组合。该符号的使用方式如示例脚本12-8.sh所示。

```
#示例脚本12-8.sh 模式范围以及范围组合的使用
#!/bin/bash

#hello123
#Hello123
#Hao123hao123
#HA0123HA0o123
#HELlo123

echo 使用模式范围
echo '匹配#[h,H]e[l,L]'
grep -E "#[h,H]e[l,L]*" 12-8.sh
echo
echo '匹配[1-9][1-9].[A-Z]'
grep -E "[1-9][1-9].[A-Z]*" 12-8.sh
```

为示例脚本12-8.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-8.sh
ben@ben-laptop:~$ ./ 12-8.sh
使用模式范围
匹配#[h,H]e[l,L]
#hello123
#Hello123

匹配[1-9][1-9].[A-Z]
#hello123
#Hello123
#Hao123hao123
#HA0123HA0o123
#HELlo123
grep -E "#[h,H]e[l,L]*" 12-8.sh
grep -E "[1-9][1-9].[A-Z]*" 12-8.sh
```

从示例脚本12-8.sh的执行结果中可以看出，grep命令可以在一定范围内进行模糊查找，如[1-9]表示从1到9的所有数字都可以匹配成功。这样可以简化匹配模板的编写，增强脚本的可读性。

12.2.2 定位符的使用

在正则表达式中，脱字符“^”和美元符号“\$”一般用来匹配行首字符和行尾字符。使用grep命令时，也可以通过这两个字符对行首或行尾进行文本的检索。在使用定位符时，还可以利用这两个字符的特性来匹配

空行。在匹配空行时，将脱字符和美元符号放在一起使用，从而使得行首和行尾之间没有其他的字符，就能匹配一个空行。定位符的使用方式如示例脚本12-9.sh所示。

```
#示例脚本12-9.sh 定位符的使用
#!/bin/bash

echo 使用脱字符匹配开头字符
grep -E "[^a-zA-Z]" test.txt

echo 使用定位符匹配空行
grep '^$' 12-9.sh

echo 使用美元符号匹配结尾字符
grep "[1-9\$].sh" 12-9.sh
```

为示例脚本12-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-9.sh
ben@ben-laptop:~$ ./ 12-9.sh
使用脱字符匹配开头字符
#hello123
#Hello123
使用定位符匹配空行
```

使用美元符号匹配结尾字符

```
grep '^$' 12-9.sh  
grep "[1-9$].sh" 12-9.sh
```

从示例脚本12-9.sh的执行结果中可以看出，在使用美元符号时，开头字符是小写字符的都能匹配成功，而使用脱字符时，结尾字符sh的前面是数字才能匹配成功。

12.2.3 字符匹配

在使用grep命令时，可以使用正则表达式中的“通配符”来表示一些无关紧要的字符，仅保留需要匹配的字符。星号“*”除了可以匹配多个文件以外，还可以用来匹配零个或多个字符。而点号一般用来匹配单个字符，这个字符可以是任意的字符。关于匹配字符的使用方式如示例脚本12-10.sh所示。

```
#示例脚本12-10.sh 字符的匹配  
#!/bin/bash  
  
#hello  
#Hello  
#HELLO  
#Hello  
#HELlo
```

```
echo 使用通配符点号匹配一个字符
```

```
grep -E "#HE.lo" 12-10.sh
```

```
echo
```

```
echo 使用通配符星号*匹配0个字符或多个字符
```

```
echo '匹配#H*'
```

```
grep -E "#H*" 12-10.sh
```

```
echo
```

```
echo '匹配*lo'
```

```
grep -E "*lo" 12-10.sh
```

为示例脚本12-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-10.sh
```

```
ben@ben-laptop:~$ ./ 12-10.sh
```

使用通配符点号匹配一个字符

```
#Hello
```

```
#HELlo
```

```
grep -E "#HE.lo" 12-10.sh
```

使用通配符星号*匹配0个字符或多个字符

匹配#H*

```
#! /bin/bash
```

```
#hello
```

```
#Hello
```

```
#HELLO
```

```
#HEllo  
#HEllo  
grep -E "#HE.lo" 12-10.sh  
echo '匹配#H*'  
grep -E "#H*" 12-10.sh  
  
匹配*lo  
#hello  
#Hello  
#HEllo  
#HEllo  
grep -E "#HE.lo" 12-10.sh  
echo '匹配*lo'  
grep -E "*lo" 12-10.sh
```

从示例脚本12-10.sh的执行结果中可以看出，在使用星号进行字符串的匹配时，凡是开头字符是#H的，都属于匹配成功。当使用点号时，只能使用点号来替换一个字符，其他的字符还需要按照实际的模板来进行匹配。

12.2.4 模式出现几率

前面介绍的grep命令在进行文本的匹配时，只要是出现一次模板中的内容，就匹配成功。因此，在正则表达式中，还可以使用大括号“{}”来限定某些字符出现的次数。只有该字符出现的次数匹配成功，

那么整个匹配才算成功。符号“{}”的使用实例如示例脚本12-11.sh所示。

```
#示例脚本12-11.sh 模式出现几率的使用
#!/bin/bash

#hello
#Hello
#helloHello

echo 匹配字符l至少出现2次
grep "l\{2\}" 12-11.sh
echo
echo 匹配字符l出现1~2次
grep "l\{1,2\}" 12-11.sh
```

为示例脚本12-11.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-11.sh
ben@ben-laptop:~$ ./ 12-11.sh
匹配字符l至少出现2次
#hello
#Hello
#helloHello

匹配字符l出现1~2次
```

```
#hello  
#HEllo  
#helloHEllo  
echo 匹配字符l至少出现2次  
grep "l\{2\}" 12-11.sh  
echo 匹配字符l出现1~2次  
grep "l\{1,2\}" 12-11.sh
```

从示例脚本12-11.sh的执行结果中可以看出，在匹配字符l至少出现2次时，在进行匹配时，那些l字符在文本中出现2次以上的才算匹配成功，而出现1到2次的那些记录，则是字符l出现了1次或2次的才算匹配成功。

注意

符号“{}”在正则表达式中属于特殊符号，因此需要使用转义字符将符号“{}”转义成普通字符。

12.2.5 匹配特殊字符

在正则表达式中使用的特殊字符也可以进行匹配，但是需要使用转义字符“\”将特殊字符进行转义，然后特殊字符就可以按照普通的字符进行匹配了。在使用grep命令进行文本的检索时，将会把特殊字符按照普通字符进行匹配。特殊字符的匹配使用实例如示例脚本12-12.sh所示。

```
#示例脚本12-12.sh 匹配特殊字符
#!/bin/bash

echo 匹配特殊字符
echo '匹配$'
grep *\$* 12-12.sh
```

为示例脚本12-12.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-12.sh
ben@ben-laptop:~$ ./ 12-12.sh
匹配特殊字符
匹配$
grep *\$* 12-12.sh
```

在匹配特殊字符“\$”时，需要使用转义字符将特殊字符转义成普通字符，这样就能够按照正常的字符进行匹配了。对于其他的字符来说，则不需要使用转义字符进行转义。

12.3 grep命令的扩展使用

grep命令除了正则表达式之外，还可以使用其他形式，如使用国际模式匹配的类名来代替正则表达式或使用egrep或fgrep命令来实现某一特殊的功能。本节将介绍grep命令的扩展使用。

12.3.1 类名的使用

在使用grep命令时，除了使用正则表达式之外，还可以使用国际模式匹配的类名来代替正则表达式。国际模式匹配的类名将正则表达式中的一些常用字符进行了“封装”，使用特定的字符进行表示，从而在使用时更加方便。常用的类名和正则表达式的关系如表12-2所示。

表12-2 国际模式匹配的类名

国际模式匹配的类名	对应的正则表达式	作用
[:upper:]	[A-Z]	匹配大写字符
[:lower:]	[a-z]	匹配小写字符
[:digit:]	[0-9]	匹配数字
[:alnum:]	[0-9a-zA-Z]	匹配字符和数字
[:space:]	空格或TAB	匹配空格
[:alpha:]	[a-zA-Z]	匹配字符，包括大写字符和小写字符

通过表12-2可以看出，国际模式匹配的类名主要是“封装”了大写字符、小写字符、数字字符、空格或tab以及大写字符和小写字符的组合、大小写字符和数字字符的组合等常用的组合形式。而在进行文本的匹配时，可以直接使用匹配的类名来代替对应的正则表达式，从而实现目标文本的检索。关于国际模式匹配类名的使用方式如示例脚本12-

13.sh所示。

```
#示例脚本12-13.sh 国际模式匹配的类名的使用
#!/bin/bash

#hello123 #Hello123
#Hao123hao123
#HA0123HA0o123
#HELlo123

echo 使用国际模式匹配的类名
echo 匹配开头字符是大写字符
grep "#[:upper:]" 12-13.sh
echo

echo 匹配开头字符是H, 后面必须是两个字符, 然后是数字
grep "H[:alpha:][[:alpha:]][[:digit:]]*" 12-13.sh
```

为示例脚本12-13.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-13.sh
ben@ben-laptop:~$ ./ 12-13.sh
使用国际模式匹配的类名
匹配开头字符是大写字符
#Hello123
#Hao123hao123
```

```
#HA0123HA0o123  
#HELLo123
```

匹配开头字符是H，后面必须是两个字符，然后是数字

```
#Hello123  
#Hao123hao123  
#HA0123HA0o123  
#HELLo123
```

从示例脚本12-13.sh的执行结果中可以看出，使用类名[:upper:]可以用来表示大写字符，grep命令会在该位置上匹配所有的大写字符，如果是大写字符则表示匹配成功。对于 [:alpha:]和[:digit:]来说，效果都是相同的。

12.3.2 egrep命令的使用

egrep命令一般用于搜索一个或多个文件，可以任意搜索文件中的字符串和符号，该命令的一般格式如下：

```
egrep [选项] [匹配模板] [查找文件列表]
```

egrep命令的使用方式和grep命令类似，都需要指定查找文件和匹配的模板，然后在文件中查找能够符合匹配模板的内容。如果查找成功，则在屏幕上输出内容。如果找不到模板中的内容，那么就没有任何显示。egrep命令还存在一些grep命令没有的选项，这些选项如表12-3所

示。

表12-3 egrep命令的特殊选项

选项/符号	作用	实例
加号+	匹配一个或多个字符	'a+[a-z]+[0-9]'
(()	匹配任意字符串	(root ubntu)

通过表12-3可以看出，使用egrep命令在某些方面比grep命令更加适合在文本中进行查找，并且非常直观，给人以清晰的认知。egrep命令的使用方式如示例脚本12-14.sh所示。

```
#示例脚本12-14.sh  egrep命令的使用
#!/bin/bash

#hello123
#Hello123
#Hao123hao123
# HA0123HA0o123
#HELlo123

echo "egrep命令的使用实例"
echo "使用加号连接字符串"
egrep 'h+[h, H]' 12-14.sh
egrep '*+123*' 12-14.sh
echo "使用grep命令实现相同的效果"
```

```
grep 'h[h, H]' 12-14.sh  
egrep '*123*' 12-14.sh  
echo "使用egrep命令匹配任意字符串"  
egrep '(h|H)(a|A)o' 12-14.sh
```

为示例脚本12-14.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-14.sh  
ben@ben-laptop:~$ ./ 12-14.sh  
egrep命令的使用实例  
使用加号连接字符串  
egrep "h+[h,H]" 12-14.sh  
grep "h[h, H]" 12-14.sh
```

使用grep命令实现相同的效果

```
egrep "h+[h,H]" 12-14.sh  
grep "h[h, H]" 12-14.sh
```

使用egrep命令匹配任意字符串

```
#Hao123hao123
```

从示例脚本12-14.sh的执行结果中可以看出，在使用egrep命令时，能够使用加号来实现几个字符串的关联匹配，而使用grep也同样可以实现，但是程序的可读性相对较差，但是就同样性来说，笔者推荐使用grep命令。

12.3.3 fgrep命令的使用

在Linux系统中，和grep命令相关的还有fgrep命令。fgrep命令常用于为文件搜索字符串。一般将模式当做固定字符串来处理，因此其处理速度很快，但是其搜索功能与grep相比，相对较弱。fgrep命令的基本使用方式如下所示。

```
fgrep [选项] [匹配模板] [查找文件列表]
```

fgrep命令搜索文件列表中符合匹配模板的数据行，如果不指定文件列表，默认从标准输出中获取数据。fgrep命令的使用方式和grep加-f选项类似，但是对于结果的处理不一样。fgrep命令如果找到符合要求的数据，那么该命令就会返回0；如果未找到匹配的内容，其返回值为1。如果返回值是大于1的数，那么该命令在执行过程中会发生语法错误，或者是查找的目标文件不存在，需要对命令重新编写。

在使用fgrep命令时，还需要将每行输入数据限制在 2048 个字节。一个段落一般限制在5000个字符的长度，并且部分选项不能重叠使用，否则会引起标志位的覆盖。关于fgrep命令的使用方式如示例脚本12-15.sh所示。

```
#示例脚本12-15.sh  fgrep命令的使用
#!/bin/bash

#hello123
#Hello123
```

```
#Hao123hao123  
# HA0123HAo123  
#HEllo123  
  
echo "fgrep命令的简单使用实例"  
fgrep "hao" *.h
```

为示例脚本12-15.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-15.sh  
ben@ben-laptop:~$ ./ 12-15.sh  
fgrep命令的简单使用实例  
#Hao123hao123  
fgrep "hao" 12-15.sh
```

在使用fgrep时数据限制在每行2048个字节，而在本例中没有达到该限制，所以，命令fgrep运行以后匹配的内容和使用grep命令是相同的。

注意

在fgrep命令中，匹配的模板都被看做是字符串，因此，所有的元字符都作为普通字符来处理。

12.4 grep命令使用实例

grep命令是Linux系统中功能非常强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。而且可以实现多种形式的搜索以及显示方式，本章将使用一些简单常用的实例来介绍grep命令的使用方式。

12.4.1 目录搜索——查找特定目录或文字

在日常的工作和学习中，经常需要查找某些文件放到哪个目录中，或是在文件中存在哪些内容。如果仅查找文件的位置，可以使用find命令。而如果需要查找文件中是否包含需要的内容，那么就要使用grep命令+正则表达式进行查找。在查找特定的文件或文字时，一般与其他的命令一起使用，或者多个grep命令一起使用，从而获取最终的目标文件或目标文字。关于目录的搜索实例如示例脚本12-16.sh所示。

```
#示例脚本12-16.sh 目录的搜索使用实例
#!/bin/bash

echo 在当前目录文件中查找文件12-16.sh
ls -l | grep 12-16.sh
echo

echo '当前文件中查找grep字符'
ls -l | grep 12-16.sh |grep "grep"
```

为示例脚本12-16.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-16.sh  
ben@ben-laptop:~$ ./ 12-16.sh  
在当前目录文件中查找文件12-16.sh  
-rwxrwxrwx 1 ben ben 152 2014-03-18 20:00 12-16.sh
```

当前文件中查找grep字符

```
ls -l | grep 12-16.sh  
echo '当前文件中查找grep字符'  
grep "grep" 12-16.sh
```

从示例脚本12-16.sh的执行结果中可以看出，在使用grep命令时，可以实现递进式的查找匹配。在每一层之间需要使用管道符进行分割。

12.4.2 使用ps命令检索特定的进程

在使用Linux系统时，运行命令以后，在系统中就会创建一个进程，当命令运行结束以后，该进程就会消失。因此，可以通过检索是否存在该进程来判断命令的执行状态。查看进程状态一般使用ps命令，并使用-ef选项，在显示所有的进程以及进程的相关信息之后，再使用grep命令检索显示出来的结果，在中间可以使用管道符将ps命令和grep命令连接在一起。如果一次不能检索出需要的进程信息，那么可以使用多个grep命令和管道符结合的方式，最终获取需要的内容，如示例脚本12-

17.sh所示。

```
#示例脚本12-17.sh 检索特定进程  
#! /bin/bash  
  
echo "进程查找"  
ps -ef | grep *i*  
ps -ef | grep i*  
ps -ef | grep init
```

为示例脚本12-17.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 12-17.sh  
ben@ben-laptop:~$ ./ 12-17.sh  
进程查找  
查找bash进程  
ben 3026 3024 0 20:06 pts/0 00:00:00 grep *bash  
  
查找12-17.sh  
ben 3024 1859 0 20:06 pts/0 00:00:00 /bin/bash ben  
ben 3028 3024 0 20:06 pts/0 00:00:00 grep 12-17.sh
```

从示例脚本12-17.sh的执行结果中可以看出，使用grep命令可以方便地找到需要的进程，并且可以通过ps命令的选项来显示关于进程的信息。

12.5 小结

本章主要介绍了Linux系统中常用的文本检索工具——grep命令。grep命令可以和正则表达式一起使用，从而形成强大的文本搜索功能。

grep命令可以使用命令选项来实现不同的输出。使用相应的选项可以实现忽略字符的大小写、只显示匹配的行数或显示不匹配的内容等，甚至还可以只显示匹配的行数，而不显示匹配的内容。

grep命令常与正则表达式一起使用来完成文本的检索。使用正则表达式可以在一定的范围内进行查找文本，使用定位符可以实现在行首或行尾进行查找，而使用通配符星号可以匹配任意个字符，点号可以匹配一个任意字符。在需要匹配特殊字符时，还可以使用转义字符将特殊字符转义成普通字符。

正则表达式还可以使用国际模式匹配的类名来代替需要匹配的字符、数字、空格等，还可以使用egrep命令来代替grep -E选项实现类似于逻辑与以及逻辑或的功能，从而方便查询多个关键字。

第13章 sed编程

一般而言，Shell脚本的主要任务之一就是处理文本文件，而使用Shell命令可以方便地对文件本身进行处理，但是对于文件内容来说，Shell命令处理起来非常麻烦。在Linux系统中，处理文件内容可以使用gedit、vi（vim）、emacs等文本编辑器，但是有些操作不需要使用这些编辑器就可以实现，此时可以使用sed编辑器和gawk编辑器。而本章将着重介绍sed工具的使用。

本章的主要内容包括：

- sed基本知识介绍
- sed的使用
- sed使用实例

13.1 认识sed

在Linux系统中，一般存在两种文本编辑器，即流编辑器和交互式文本编辑器。gedit、vi（vim）、Emacs等属于文本编辑器，文本编辑器可以使用键盘对文本文件进行添加、删除、更改等操作，实现交互式操纵。而流编辑器则是使用预先定义的操作规则对文本文件进行添加、删除、更改等操作。

13.1.1 sed工作模式

sed编辑器以行为单位进行文本的处理，常用于以批处理方式编辑文件，如自动编辑或处理一个或多个文件等。sed编辑器的基本使用方式如下：

```
sed [选项] 命令 目标文件  
sed [选项] -f 脚本文件 目标文件
```

上面的表达式中展示了sed的两种使用方式，目标文件可以是一个文件，也可以是多个文件。第1种方式是使用sed内置的命令进行目标文本文件的操作，使用命令可以实现文本文件的各种操作。在使用该命令时，需要用一对单引号把命令括起来，防止被看做是普通的选项或其他的内容。第2种方式是使用定义好的脚本文件，对目标文本文件进行操作。这两种方式都可以实现对目标文本文件的处理。

sed编辑器可以根据输入命令行的命令或者存储在命令文本文件中的命令处理数据。它每次从输入读取一行数据，将该数据与所提供的编辑器命令进行匹配，根据命令修改数据流中的数据，然后将新数据输出到STDOUT。在流编辑器将全部命令和一行数据匹配完之后，读取下一行数据，并重复上述过程。处理完数据流中的全部数据行之后，流编辑器停止工作。sed编辑器的处理过程如图13-1所示。

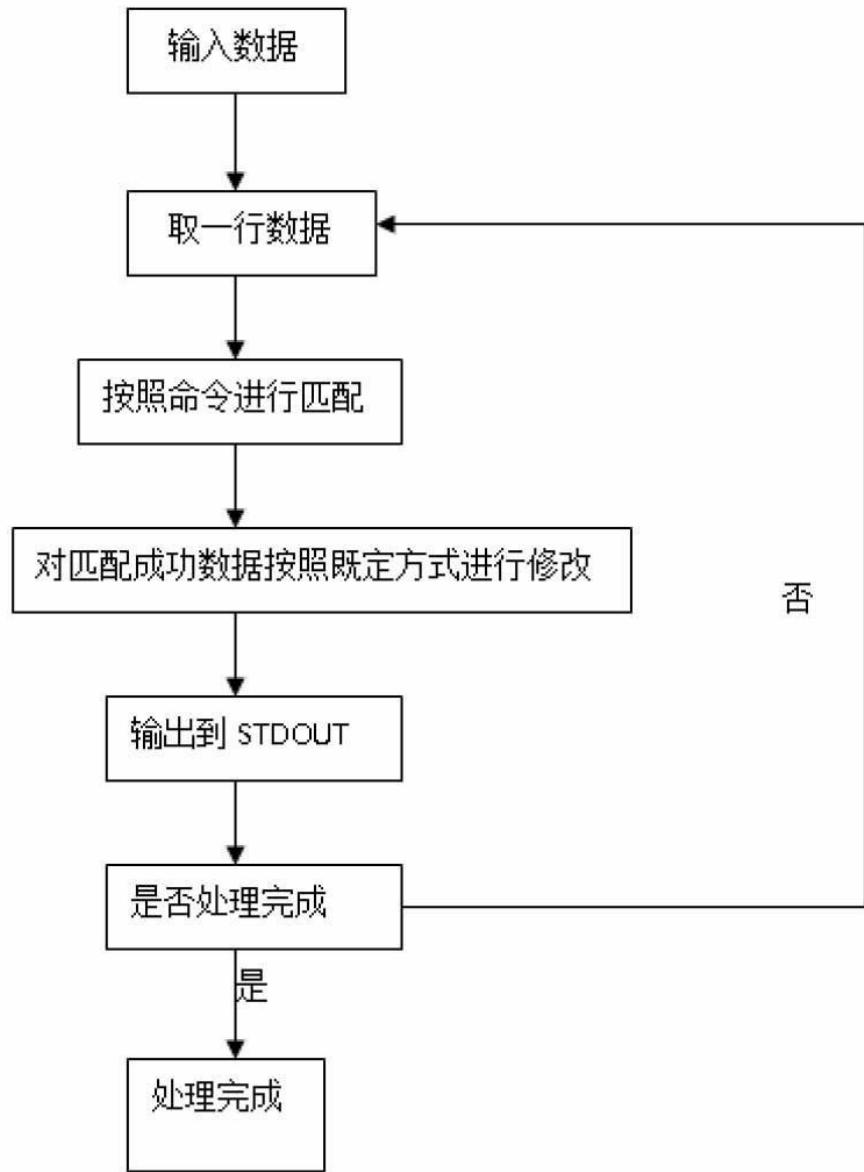


图13-1 sed编辑器运行过程

13.1.2 sed常用指令

sed编辑器所实现的很大一部分功能是由一些命令来实现的，这些命令告诉sed编辑器将要执行什么操作，以及在什么范围内进行这些操

作。Sed编辑器常用的命令如表13-1所示。

表13-1 sed编辑器常用命令

命令	作用
a\	在当前行后面追加文本
b	分支到脚本中带有标记的地方，如果分支不存在则分支到脚本的末尾
d	删除命令
c	更改选定的行，并且替换整行内容
D	删除第一行
i\	在当前行上面插入文本
h	将文本中的内容放到内存中的缓冲区
H	以追加的方式将文本中的内容放到内存缓冲区
g	获得内存缓冲区的内容，并替代当前模板块中的文本
G	获得内存缓冲区的内容，并追加到当前模板块文本的后面
l	列表不能打印字符的清单
n	读取下一个输入行，用下一个命令处理新的行而不是用第一个命令
N	追加下一个输入行到模板块后面并在二者间嵌入一个新行，改变当前行号

p	打印文本文件的指定行
P (大写)	打印模板块的第一行
q	退出sed
r file	从file中读行
s	替换命令，一般用来替换字符串
w file	写并追加模板块到file末尾
W (大写) file	写并追加模板块的第一行到file末尾
!	反向选择，匹配命令没有发生作用的行

通过表13-1可以看出，使用sed命令能够实现字符串的替换、删除等常用操作，这些命令的使用方式和选项类似，需要和sed命令一起使用。接下来就分别介绍表13-1中常用命令的使用方式。

13.1.3 sed常用选项

sed编辑器除了常和命令一起使用之外，还可以使用选项。这些选项和其他的Shell命令一样，可以丰富sed编辑器的处理方式。常用的选项如表13-2所示。

表13-2 sed编辑器常用选项

选项	作用
-e	允许多态编辑

-h	帮助选项
-n	取消默认输出
-f	引导sed命令文件
-V	打印版本信息和版权信息

通过表13-2可以看出，使用sed命令的选项可以显示sed编辑器的某些基本信息，如显示帮助信息、版权信息等。还可以对输入输出进行某些设定，如取消默认输出、引导sed命令文件等。关于这些选项的使用方式如示例脚本13-1.sh所示。

```
#示例脚本13-1.sh sed常用选项的使用
#!/bin/bash

echo 取消默认输出
sed -n '/a/p' data.txt
echo

echo 使用默认输出
sed '/a/p' data.txt
```

为示例脚本13-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-1.sh
ben@ben-laptop:~$ ./ 13-1.sh
取消默认输出
```

```
this is a
```

使用默认输出

```
this is a
```

```
this is a
```

```
this is b
```

```
this is c
```

通过示例脚本13-1.sh的执行结果可以看出，在使用默认输出和不使用默认输出时，sed编辑器同样的命令执行后输出结果是不一样的。因此，在实际操作时，要明白是不是需要默认输出。否则即使命令使用正确，显示结果也是“错误”的。

13.1.4 sed地址范围

在默认情况下，sed编辑器是对所有的文本进行各种相应的操作。然后sed编辑器还可以在一定的范围内进行文本的操作，如仅删除第3行的文本，而对其他文本不进行任何操作，或者对中间的某几行进行操作。在Bash中，sed编辑器中可以使用的操作地址范围一般用一些符号表示。这些符号如下所示。

- 数字定位：定义地址范围。
- 符号\$：最后一行。
- 匹配字符串：处理存在匹配字符串的行。

Sed编辑器的地址范围可以有很多表达方式。可以只处理其中的某

几行，或者直接定位到最后一行，还可以同时使用数字和最后一行，以表示控制从某一行到最后一行，甚至还可以单独处理存在某些字符串的行。关于sed编辑器地址范围符号的使用方式如示例脚本13-2.sh所示。

```
#date1.txt文本内容
this is a
this is b
this is c
this is d

#示例脚本13-2.sh sed地址范围符号的使用
#!/bin/bash

echo 删除第3行到末尾的所有内容
sed '3,$d' data1.txt
echo

echo 删除第2行到第3行的内容
sed '2,3d' date1.txt
echo

echo 不使用地址范围符号
sed 'd' date1.txt
```

为示例脚本13-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-2.sh
```

```
ben@ben-laptop:~$ ./ 13-2.sh
```

删除第3行到末尾的所有内容

```
this is a
```

```
this is b
```

删除第2行到第3行的内容

```
this is a
```

```
this is d
```

不使用地址范围符号

通过示例脚本13-2.sh的执行结果可以看出，在使用地址范围之后，`sed`编辑器只处理确定地址范围的文本，而对确定范围之外的文本不进行任何的处理。而在默认的处理方式中，`sed`编辑器对所有的数据都进行处理。

13.2 `sed`编辑器常用命令

`sed`编辑器对于Shell脚本来说是常用的工具之一，而使用`sed`编辑器的基础就是熟练使用`sed`编辑器中常用的命令，如文本的替换、删除以及添加等操作，本节将重点介绍这些命令的使用方式。

13.2.1 替换命令的使用

`sed`命令中比较常用的命令之一就是替换命令，即使用某些字符串

来替换文本中特定的字符串，其使用方式如示例脚本13-3.sh所示。

```
ben@ben-laptop:~$cat data.txt
this is a
this is b
this is c
#示例脚本13-3.sh 替换命令的使用
#!/bin/bash

echo 'sed替换命令的基本使用'
echo 替换this
sed 's>this/sed' data.txt
echo

echo 替换is
sed 's/is/sed'data.txt
```

为示例脚本13-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-3.sh
ben@ben-laptop:~$./13-3.sh
sed替换命令的基本使用
替换this
test is a
test is b
test is c
```

```
替换is  
thtest is a  
thtest is b  
thtest is c
```

通过示例脚本13-3.sh的执行结果可以看出，使用了替换命令s之后，在文本中的实例this和is都被替换成文本test。然而在默认情况下，sed命令只会替换文本中第一次出现待替换的文本，而对后面的文本不进行处理，此时，需要在sed命令中添加替换标记，从而实现所有目标文本的替换。常用的替换标记如下。

- 数字：表示替换文本中的第几个实例。
- w file：表示将替换的结果写入到文件file。
- g：表示用新文本替换现有文本中的全部实例。
- p：表示打印原始行的内容。

在使用替换标记时，要将标记放在替换文本的后面，其基本格式如下：

```
命令s/新文本/实例/替换标记
```

在使用替换标记以后，sed编辑器就会按照特定的替换方式进行文本的替换，如示例脚本13-4.sh所示。

```
#示例脚本13-4.sh 替换标记的使用  
#!/bin/bash
```

```
echo 替换第2个is  
sed 's/is/sed/2' data.txt  
echo  
  
echo 替换全部的is  
sed 's/is/sed/g' data.txt
```

为示例脚本13-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-4.sh  
ben@ben-laptop:~$ ./13-4.sh  
替换第2个is  
this test a  
this test b  
this test c  
  
替换全部的is  
thtest test a  
thtest test b  
thtest test c
```

通过示例脚本13-4.sh的执行结果可以看出，在使用了替换标记以后，将按照替换标记来进行文本的替换，而不会采用默认的方式只替换第一个实例。

13.2.2 删除命令的使用

在使用sed编辑器的时候，常用的命令还有文本的删除命令，也就是将文本中的目标实例从文本中删除。删除命令在使用时需要放到文本的后面，从而删除在文本中出现实例文本所在的整行。删除命令的操作方式如示例脚本13-5.sh所示。

```
#示例脚本13-5.sh    删除命令的使用
#!/bin/bash

echo 删除is所在行
sed '/is/d' data.txt
echo

echo 删除is a所在行
sed '/is/d' data.txt
echo

echo 查看文件data.txt的内容
cat data.txt
```

为示例脚本13-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-5.sh
ben@ben-laptop:~$ ./13-5.sh
删除is所在行
```

```
删除is a所在行
```

```
this is b
```

```
this is c
```

```
查看文件data.txt的内容
```

```
this is a
```

```
this is b
```

```
this is c
```

通过示例脚本13-5.sh的执行结果可以看出，在删除is所在行的时候，所有包含is的文本全部被删除。而在删除操作之后，源文件的内容并没有发生变化，发生变化的只是sed编辑器的输出内容。但是在使用删除命令的时候还是需要谨慎。

注意

在进行删除操作时，`sed`默认的操作范围是所有的文本，因此最好要和地址范围一起使用。

13.2.3 文本的添加和替换

`sed`编辑器还可以对文本进行整行的添加和替换，在添加数据时，还可以确定是在目标行之前添加，还是之后添加，这两部分涉及的命令如下。

- 插入命令*i*: 在指定行之前添加新的一行。
- 附加命令*a*: 在指定行之后添加新的一行。
- 更改命令*c*: 使用新的数据替换指定行。

上述命令的基本使用格式如下：

操作地址+操作命令/新文本

在上面的格式中，操作地址和操作命令可以放在一块，而不需要使用加号进行连接。在使用上述命令时，可以对所有的目标行进行操作，也可以只对特定的行进行操作，操作方式如示例脚本13-6.sh所示。

```
#示例脚本13-6.sh 文本的添加和替换
#!/bin/bash

echo 在第2行后面追加一行
sed '3a\this is add line ' data.txt
echo

echo 在第2行前面追加一行
sed '3i\this is add line ' data.txt
echo

echo 替换第2行
sed '3c\this is add line ' data.txt
```

为示例脚本13-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-6.sh
```

```
ben@ben-laptop:~$ ./13-6.sh
```

在第2行后面追加一行

```
this is a
```

```
this is b
```

```
this is add line
```

```
this is c
```

在第2行前面追加一行

```
this is a
```

```
this is add line
```

```
this is b
```

```
this is c
```

替换第2行

```
this is a
```

```
this is add line
```

```
this is c
```

通过示例脚本13-6.sh的执行结果可以看出，使用sed编辑器可以实现文本的添加、替换等操作，可以在当某些文件中缺少内容或需要进行文本的替换时使用这些命令。

13.3 高级sed编程

在前面介绍了sed编辑器中常用的、简单的操作方式，一般情况

下，掌握上面的操作就可以了。但是在某些特殊情况下，还需要了解sed编辑器的一些高级用法，这样就可以更加深刻地了解sed编辑的操作方式和使用方法。

13.3.1 同时处理多行数据

在前面的操作中，sed编辑器每次只处理一条数据，然而在有些时候，需要同时处理多条记录，如在进行替换时，某些数据可能会分行显示，如果只处理一行数据，那么这些“特殊”的实例就不会被处理，此时就需要同时处理多行数据。在sed编辑器中，用于多行处理的命令如下。

- n：移动到下一行文本。
- N：创建多行组。
- D：删除多行组中的单个行。
- P：打印多行组中的单个行。

命令n是next的缩写，可以使sed编辑器在处理文本时，直接跳转到文本的下一行，而不是等sed编辑器处理完所有的命令以后才回到下一行。

上面介绍的命令n用于处理单行文本，而对于多行数据的处理来说，大写的N命令是将两行数据合并为一行数据，从而方便处理。在使用N命令之后，可以使用D命令删除多行或使用P命令进行数据的打印。这几个命令的使用方式如示例脚本13-7.sh所示。

```
ben@ben-laptop:~$cat data.txt
```

```
this  
is a  
this  
is b  
this is c  
ben@ben-laptop:~$cat 13-7.sh  
#示例脚本13-7.sh 使用多行命令  
#!/bin/bash  
sed '  
> N  
> /this\nis/d  
> ' data.txt
```

为示例脚本13-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-7.sh  
ben@ben-laptop:~$./13-7.sh  
this  
is a  
this  
is b
```

通过示例脚本13-7.sh的执行结果可以看出，在脚本中将分行的this和is进行了合并，然后进行了删除操作，而命令执行后的输出结果也是按照两行输出的，从而将符合条件的记录删除。

13.3.2 sed编辑器的空间

对于sed编辑器来说，不直接去改变处理文本的内容，而是将处理结果保存到一个缓冲区中，该缓冲区称为sed编辑器的模式空间。该空间是一个活动的缓冲区，在sed编辑器处理命令时保留被检查的文本。而sed编辑器还可以在处理模式空间中的其他行时，使用保留空间暂时保留文本行。在sed编辑器中，可以使用特殊的命令对保留空间中的内容进行操作。对于这两种空间常用的操作命令如表13-3所示。

表13-3 空间操作相关命令

命令	作用
h	将模式空间复制到保留空间
H	将模式空间追加到保留空间
g	将保留空间复制到模式空间
G	将保留空间追加到模式空间
x	将保留空间和模式空间内容互换

表13-3中命令的一般使用方式如下面的操作所示：

```
ben@ben-laptop:~$cat data.txt
this is a
this is b
this is c
```

```
ben@ben-laptop:~$cat 13-8.sh
#示例脚本13-8.sh 使用多行命令
#!/bin/bash

sed -n '/b/{> h
> p
> n
> p
> g
> p
> }' data.txt
```

为示例脚本13-8.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-8.sh
ben@ben-laptop:~$./13-8.sh
this is b
this is c
this is b
```

通过示例脚本13-8.sh的执行结果可以看出，`sed`编辑器两个空间中的数据可以进行各种切换，而最终的数据需要保存到模式空间中。示例脚本13-8.sh的执行过程如图13-2所示。

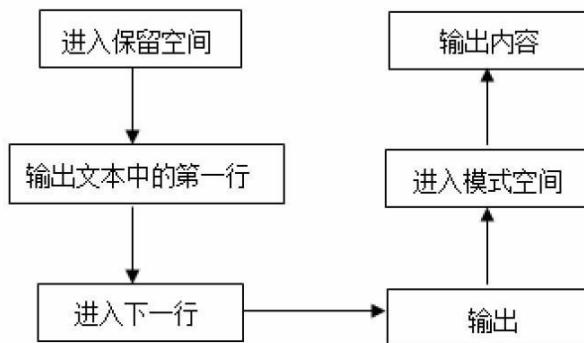


图13-2 sed空间切换示意图

注意

对于一般情况来说，使用h、H等命令将数据移动到保留空间之后，还需要使用g、G命令移动到模式空间，从而保持数据的一致性。

13.3.3 sed编辑器的反向

在使用sed编辑器进行实际操作时，有时会出现如果使用反向操作会更加简单明了的情形，如对文本中不包含is的语句进行操作等。此时就需要使用感叹号命令实现反向操作。

感叹号（!）命令的作用就是反向操作，原本可以进行操作的情形，在使用了感叹号命令时不进行操作，如示例脚本13-9.sh所示。

```
ben@ben-laptop:~$cat data.txt
this is c
this is b
```

```
this is d  
ben@ben-laptop:~$cat 13-9.sh  
#示例脚本13-9.sh 反向命令  
#!/bin/bash  
  
echo 输出没有a的行  
sed -n '/a/!p' data.txt
```

为示例脚本13-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-9.sh  
ben@ben-laptop:~$./13-9.sh  
输出没有a的行  
this is c  
this is b  
this is d
```

通过示例脚本13-9.sh的执行结果可以看出，使用了反向命令之后，以前满足条件的结果不再输出，转而输出的是不满足条件的结果。

13.3.4 重定向**sed**的输出

在默认情况下，**sed**编辑器的输出结果是展示在STDOUT上的，而原文本没有任何的变化。在某些情况下，需要对**sed**编辑器的输出结果进行处理，此时就需要对**sed**编辑器的输出进行重定向，而不是仅在

STDOUT上进行展示。

sed编辑器的输出一般重定向到某个变量中，从而方便在脚本中使用。在Shell中，一般使用反引号实现该功能，如示例脚本13-10.sh所示。

```
ben@ben-laptop:~$cat data.txt
this is a
ben@ben-laptop:~$cat 13-10.sh
#示例脚本13-10.sh 重定向sed输出
#!/bin/bash

echo 替换this不使用重定向
sed 's>this/sed'data.txt
echo

echo 使用重定向
NewResult=`sed 's>this/sed'data.txt`
echo $NewResult
```

为示例脚本13-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 13-10.sh
ben@ben-laptop:~$./13-10.sh
替换this不使用重定向
sed is a
```

使用重定向

```
sed is a
```

通过示例脚本13-10.sh的执行结果可以看出，在不使用重定向时，`sed`编辑器的处理结果将只显示在STDOUT中，当再次使用时将非常的麻烦。而将处理结果重定向到一个变量时，可以通过操作变量来对`sed`编辑器的结果进行处理。

注意

如果待处理文本中存在很多行，那么重定向后，存储在变量中的值仅是文本中的最后一行，如果需要对其他的数据进行处理，需要对输出结果做循环处理。

13.4 小结

本章重点介绍了`sed`编辑器的使用方式。`sed`编辑器作为一种流编辑器，可以使用预先定义的操作规则对文本文件进行添加、删除、更改等操作。

`sed`编辑器功能的实现需要与`sed`命令的选项和命令一起使用，其默认的操作方式是从文本的开头进行处理，一直处理到文本末尾。在处理过程中，可以按照特定的模式进行处理，即只处理能够匹配的文本行，还能够使用表示地址范围的命令来直接指示处理哪些数据。

`sed`编辑器的常用操作方式是对文本进行替换、删除和添加，可以

分别使用不同的命令实现，然后进行操作以后，文本中的内容并没有发生变化，发生变化的只是sed命令的输出结果。

sed除了一些常用功能之外，还有一些高级但是不常用的功能。使用sed编辑器能够同时处理多行数据，还可以使用感叹号命令进行反向操作。而sed编辑器的处理结果除了可以在STDOUT中输出之外，还可以使用反引号重定向到某个变量中，从而在后续的操作中，通过操作变量来实现对sed编辑器处理结果的操作。

第14章 gawk编程

前面的章节中介绍了sed在Shell脚本中的使用，但是sed编辑器具有一定的局限性，只能实现简单的操作方式，而不能在编辑过程中添加变量以及其他结构。在Linux系统中，还可以使用gawk编辑器来实现更加高级的处理方式。gawk编辑器类似于一个编程环境的工具，允许修改和重新组织文件中的数据。本章将重点介绍如何在Shell脚本中使用gawk工具对文本进行处理。

本章的主要内容如下：

- gawk概述
- 变量在gawk中的使用
- 各种结构在gawk中的使用
- 函数在gawk中的使用

14.1 gawk概述

gawk程序是Linux系统中awk的GNU版本，gawk程序更近似于一种编程语言，而不仅仅是一种编辑器，在gawk程序中，可以实现编程语言中常用的功能，这些功能如下：

- 使用变量；
- 使用算术操作符和字符串操作符进行运算；
- 可以使用各种结构化编程的方式。

本节将重点介绍gawk程序的基本用法。

14.1.1 gawk基本介绍

gawk程序是由命令gawk来实现的，gawk命令的基本格式如下：

```
gawk [选项] [需要执行的文件]
```

gawk常用的选项如表14-1所示。

表14-1 gawk常用选项

选项	作用
-F fs	指定描绘一行中数据字段的文件分隔符
-f file	指定读取程序的文件名
-v	定义程序中使用的变量和默认值
-mf N	指定数据文件中要处理的字段的最大数目
-mr N	指定数据文件中最大记录的大小
-W keyword	指定gawk的兼容模式或警告级别

gawk程序一般是在一对大括号中定义的，而要执行的命令也需要放置到大括号之间，并且使用单引号括起来，用来标注是需要执行的命令。而使用了选项之后，gawk命令能够运行的命令就更加丰富了。

gawk的简单使用如示例脚本14-1.sh所示。

```
#示例脚本14-1.sh gawk的简单使用
#!/bin/bash

echo "gawk的简单使用"
gawk '{print "this is first gawk program"}'
```

为示例脚本14-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-1.sh
ben@ben-laptop:~$ ./14-1.sh
gawk的简单使用
```

通过脚本14-1.sh的运行结果可以看出，当运行了该脚本以后，不会有任何的信息展示。这是因为gawk命令在命令行中没有指定文件名，gawk需要在STDIN中获取数据，而在运行脚本时未指定任何输入数据，而gawk程序一直在等待用户的输入，gawk程序没有任何的输出信息。可将脚本改动一下，如示例脚本14-2.sh所示。

```
#示例脚本14-2.sh gawk的简单使用
#!/bin/bash

echo gawk的简单使用
gawk '{print "this is first gawk program"}' 14-1.sh
```

为示例脚本14-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-2.sh
ben@ben-laptop:~$ ./14-2.sh
gawk的简单使用
this is first gawk program
```

通过示例脚本14-2.sh的执行过程可以看出，输出的内容都是一样的，执行了4次print命令。这是因为gawk命令对数据流中的每一行文本都会执既定的脚本程序，因为在本例中执行的是简单的输出语句，因此输出的内容是一样的。

注意

如果想结束gawk程序，可以使用Ctrl+D组合键生成EOF字符，从而结束gawk程序的运行。

14.1.2 gawk基本使用

gawk的主要功能和sed一样，可以按行对文件中的文本进行各种编辑和处理。而在文本文件中，字段一般会使用空格等字符进行分隔，在gawk中，一般使用字段分隔符来区分每一个字段，而为了表述方便，会

使用变量\$1、\$2.....\$n来表示每一个字段，表示方式如下：

- \$1表示每一行的第一个字段；
- \$2表示每一行的第2个字段；
- \$n表示每一行的第n个字段；
- \$0表示每一行的所有信息。

在gawk程序中，可以使用上面的这些变量来对文件中的文本进行“定点”操作，如示例脚本14-3.sh所示。

```
#示例脚本14-3.sh    获取字段
#!/bin/bash

echo 显示字段1
gawk '{print $1}' 14-3.sh
echo

echo 显示字段2
gawk '{print $2}' 14-3.sh
```

为示例脚本14-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-3.sh
ben@ben-laptop:~$ ./14-3.sh
显示字段1
#!
```

```
echo
```

```
gawk
```

```
echo
```

```
echo
```

```
gawk
```

显示字段2

```
/bin/bash
```

显示字段1

```
'{print
```

显示字段2

```
'{print
```

通过实例脚本14-3.sh的执行结果可以看出，在输出字段的时候，\$1和\$2分别表示第1个和第2个字段，而字段之间的分隔使用空格来表示。

注意

字段分隔符一般使用空字符来表示，在Linux系统中，空字符一般是空格、Tab键等。

gawk程序默认的字段分隔符是空格，而如果某个文本中的分隔符不是空格，就可以使用-F选项来执行字段分隔符，如示例脚本14-4.sh所

示。

```
#实例脚本14-4.sh    更改分隔符
#!/bin/bash

echo 指定分隔符为字符a
echo this is a gawk program
gawk -Fa '{print $1}' 14-4.sh
echo

echo 指定分隔符为字符c
gawk -Fc '{print $1}' 14-4.sh
```

为示例脚本14-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-4.sh
ben@ben-laptop:~$ ./14-4.sh
指定分隔符为字符a
this is a gawk program
#实例脚本14-4.sh    更改分隔符a
#!/bin/b

echo 指定分隔符为字符
echo this is
g
echo
```

```
echo 指定分隔符为字符c  
g  
  
指定分隔符为字符c  
#实例脚本14-4.sh    更改分隔符  
#!/bin/bash  
  
e  
e  
gawk -Fa '{print $1}' 14-4.sh  
e  
  
e  
gawk -F
```

通过脚本14-4.sh的执行结果可以看出，使用FS制定了分割符以后，文件中的字符分隔不是在按照原来的空字符进行，而是按照指定的字符进行分隔。

需要执行的命令除了在gawk程序中指定之外，还可以指定一个脚本文件，在执行gawk命令时使得执行的命令从该文件中读取，此时就需要使用-f选项来指定gawk程序从哪个文件中读取。当读取到文件以后，gawk就会按照程序中的内容执行。在示例脚本14-5.sh中，执行的内容放在14-5-1.sh中，如下所示。

```
#实例脚本14-5.sh    指定执行文件
```

```
#!/bin/bash

echo 指定程序执行的文件
gawk -f 14-5-1.sh 14-5.sh
```

为示例脚本14-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$cat 14-5-1
{print $1}
ben@ben-laptop:~$ chmod u+x 14-5.sh
ben@ben-laptop:~$./14-5.sh
指定程序执行的文件
#!echo
awk
```

通过示例脚本14-5.sh的执行结果可以看出，当使用-f选项指定了执行文件后，在执行gawk命令时，将会将使用文件中的内容来对每一行文本进行处理，其处理方式是类似的，直至将所有的文本处理完成。

14.2 变量的使用

变量一般用来存储一些临时的内容。在gawk程序中，存在内置变量和自定义变量两种形式，本节将重点介绍变量在gawk程序中的使用方

式。

14.2.1 内置变量的使用

在gawk中存在几个内置变量，可以在编写gawk程序时使用，这些内置变量类似于Linux系统中的环境变量，可以在任何的gawk程序中使用，但是在每一个程序中分别代表不同的值。这些内置变量如表14-2所示。

表14-2 gawk内置变量

变量名	作用
NR	已经读取过的记录数
FNR	从当前文件中读出的记录数
FILENAME	输入文件名
FS	字段分隔符
RS	记录分隔符
OFMT	数字的输出格式
OFS	输出字段分隔符
ORS	输出记录分隔符
NF	当前记录中的字段数

对于表14-2中的变量，如果只处理一个文件，那么变量NR和FNR的值是一样的。而当处理多个文件时，每一个文件的FNR值都不会相同，但是对于NR来说，却是每一个文件都是相同的。在默认情况下，RS为换行符，而FS为空格。使用这些变量能够非常简单地获取文件中的某些信息，如示例脚本14-6.sh所示。

```
#示例脚本14-6.sh    使用内置变量
#!/bin/bash

echo 显示文件名
gawk 'NR==1 { print FILENAME }' 14-6.sh
echo

echo 显示当前记录中的字段数
gawk '{print NF}' 14-6.sh
```

为示例脚本14-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-6.sh
ben@ben-laptop:~$ ./14-6.sh显示文件名
14-5.sh

显示当前记录中的字段数
2
2
0
```

```
2  
7  
1  
0  
2  
4  
1  
0  
2  
5  
0  
0
```

通过示例脚本14-6.sh的执行结果可以看出，内置变量可以方便地将gawk命令在处理文本时的一些信息进行保存。而在使用gawk命令时，也可直接使用这些内置变量进行各种操作。

内置变量的值并不是一成不变的，而是可以进行修改的。修改时可以直接将值赋值给相应的变量，如果想在后续的处理过程中使用新的变量值，那么只需要将表示数值变化的语句放在处理语句的前面即可，这样后续的处理语句就能够在执行过程中使用新的数值来执行，如示例脚本14-7.sh所示。

```
#示例脚本14-7.sh 修改内置变量的值  
#!/bin/bash  
  
echo 更改文件名
```

```
gawk ' { FILENAME ="14-5.sh"} { print FILENAME} '
```

为示例脚本14-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-7.sh
ben@ben-laptop:~$ ./14-7.sh
更改文件名
14-5.sh
14-5.sh
14-5.sh
14-5.sh
14-5.sh
```

通过示例脚本14-7.sh的执行结果可以看出，内置变量可以通过人工干预其结果，但是一般不会这么操作，防止在后续的操作过程中发生错误。

注意

使用gawk命令一般会修改字段分隔符FS，这样可以按照新的分隔方式来对文本进行处理。

14.2.2 自定义变量的使用

在gawk程序中，除了可以使用内置变量之外，还可以使用需要的自

定义变量。在使用自定义变量时，也不需要提前定义或声明，可以直接使用。自定义变量的赋值方式如下：

```
变量名=变量值
```

在使用自定义变量时，一般直接将变量的值赋值给变量名。变量名一般由字符、数字和下划线组成，但是开头字符一般不是数字。在定义变量时，一般不指定变量的类型，gawk会根据变量的内容自动确定变量的类型。如果变量没有初始值，那么gawk会根据变量的类型来确定变量的值。对于字符串变量来说，一般会赋值为NULL，而对于数值来说，一般赋值为0。在gawk中自定义变量的使用如示例脚本14-8.sh所示。

```
#示例脚本14-8.sh 使用自定义变量
#!/bin/bash

echo 自定义变量
gawk ' { FR="test" } { print FR } ' 14-8.sh
```

为示例脚本14-8.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-8.sh
ben@ben-laptop:~$ ./14-8.sh
自定义变量
test
test
test
```

```
test  
test
```

通过示例脚本14-8.sh的执行结果可以看出，在gawk中定义了变量FR，然后就可以在后续的操作中使用该变量，在本例中仅输出变量的值，而未进行其他的操作。在实际操作中，可以根据需要进行各种相应的操作。

14.2.3 数组的使用

在gawk中，同样可以使用数组来表示一部分数据的组合。在gawk中，数组一般被称为关联数组，这是因为数组的下标可以是数，也可以是串，并且在gawk中，数组和其他的变量一样，不需要提前定义或声明，而是可以直接使用。而对于数组元素的初始化来说，可以根据数组中表示的内容的不同而初始化为0或空字符串。在gawk中，数组的定义方式如示例脚本14-9.sh所示。

```
#示例脚本14-9.sh 数组的使用  
#!/bin/bash  
  
echo 使用数字作为数组的下标  
echo 输出数值中的元素  
gawk 'BEGIN{numarray[1]="this";  
      numarray[2]="is";  
      numarray[3]="num";}'
```

```
numarray[4]="index";
{for(i in numarray) print numarray[i]}''
echo

echo 使用字符串作为数组的下标
echo 输出数组中的元素

gawk 'BEGIN{strarray ["first"]="this";
           strarray["second"]="is";
           strarray["third"]="string";
           strarray["four"]="index";
{for(j in strarray) print strarray[j]}'
```

为示例脚本14-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-9.sh
ben@ben-laptop:~$ ./14-9.sh
输出数值中的元素
this
is
num
index
```

使用字符串作为数组的下标

输出数组中的元素

```
is
index
```

```
this  
string
```

通过示例脚本14-9.sh的执行结果可以看出，数组同样可以在gawk命令中使用，可以根据在编写脚本时的实际需要进行使用。

14.3 结构的使用

在前面的章节中已经介绍了Shell脚本中常用的结构，如条件结构、循环结构等。在gawk中，这些结构也同样可以使用。在使用这些结构时，先从目标文件中获取一条记录，然后再按照相应的结构执行，当执行完毕后，再取下一条记录。也就是说每一条记录都会单独地在相同的结构中执行一次。本节将详细介绍各种结构形式在gawk程序中的使用。

14.3.1 条件结构的使用

在第7章中介绍了条件结构的使用方式。条件结构也可以称为分支结构，一般是在满足某些条件下执行某些语句，而在满足另外的条件时，另外的语句将得到执行。在gawk程序中，条件结构也可以使用if...else结构来实现。条件结构的基本结构是if结构，该结构的基本形式如下：

```
if (逻辑表达式)  
{
```

语句

}

在上面的结构中，首先判断逻辑表达式的运算结果是真还是假，如果逻辑表达式的运算结果为真，那么if结构中的语句才会得到执行。如果逻辑表达式的运算结果为假，那么语句将不会得到执行。关于if结构的简单使用方式如示例脚本14-10.sh所示。

```
#示例脚本14-10.sh if结构的简单使用
#!/bin/bash

echo if结构在gawk中的使用
gawk '{if ($1 > $2) {print $1 "大于" $2} }' data.txt
```

为示例脚本14-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$cat data.txt
10 20
10 5
10 10
ben@ben-laptop:~$ chmod u+x 14-10.sh
ben@ben-laptop:~$./14-10.sh
if结构在gawk中的使用
10大于5
```

通过示例脚本14-10.sh的执行结果可以看出，在if结构中，如果逻辑

表达式的运算结果为假，那么是没有任何的操作的。为了提高程序的逻辑性，可以使用if...else结构来代替if结构，该结构的基本形式如下：

```
if (逻辑表达式)
```

```
{
```

```
语句1
```

```
}
```

```
else
```

```
{
```

```
语句2
```

```
}
```

在if...else结构中，如果逻辑表达式的运算结果为逻辑真，那么将执行结构中的语句1，而如果逻辑表达式的运算结果为逻辑假，那么语句2将得到执行。使用if...else结构可以避免当逻辑表达式运算结果为空时，没有特定的语句执行的问题。该结构的使用方式如示例脚本14-11.sh所示。

```
#示例脚本14-11.sh  if else结构的使用
```

```
#!/bin/bash
```

```
echo if结构在gawk中的使用
```

```
gawk '{if ($1 > $2) {print $1 "大于" $2}  
else {print $1 "不大于" $2}}'
```

```
data.txt
```

为示例脚本14-11.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$cat data.txt
10 20
10 5
10 10
ben@ben-laptop:~$ chmod u+x 14-11.sh
ben@ben-laptop:~$./14-11.sh
if结构在gawk中的使用
10不大于20
10大于5
10不大于10
```

在示例脚本14-11.sh的执行结果中，只能区分两者的大小，而如果两者相等，则无法区分。除此之外，还有很多情形需要不止一次地进行判断，此时一般会嵌套使用if...else结构，将所有可能出现的情况都进行处理，其基本形式如下：

```
if (逻辑表达式1)
```

```
{
```

```
语句1
```

```
}
```

```
else if (逻辑表达式2)
```

```
{
```

语句2

```
}
```

```
else if (逻辑表达式3)
```

```
{
```

语句3

```
}
```

```
else
```

```
{
```

语句4

```
}
```

在上面的结构中，首先判断逻辑表达式1的执行结果，如果逻辑表达式1的执行结果为逻辑真，那么就执行语句1。如果逻辑表达式1的执行结果为假，那么判断逻辑表达式2的执行结果，如果为真，则执行语句2，否则继续执行逻辑表达式3，如果所有的逻辑表达式都不成立，那么语句4将得到执行。其执行顺序如图14-1所示。

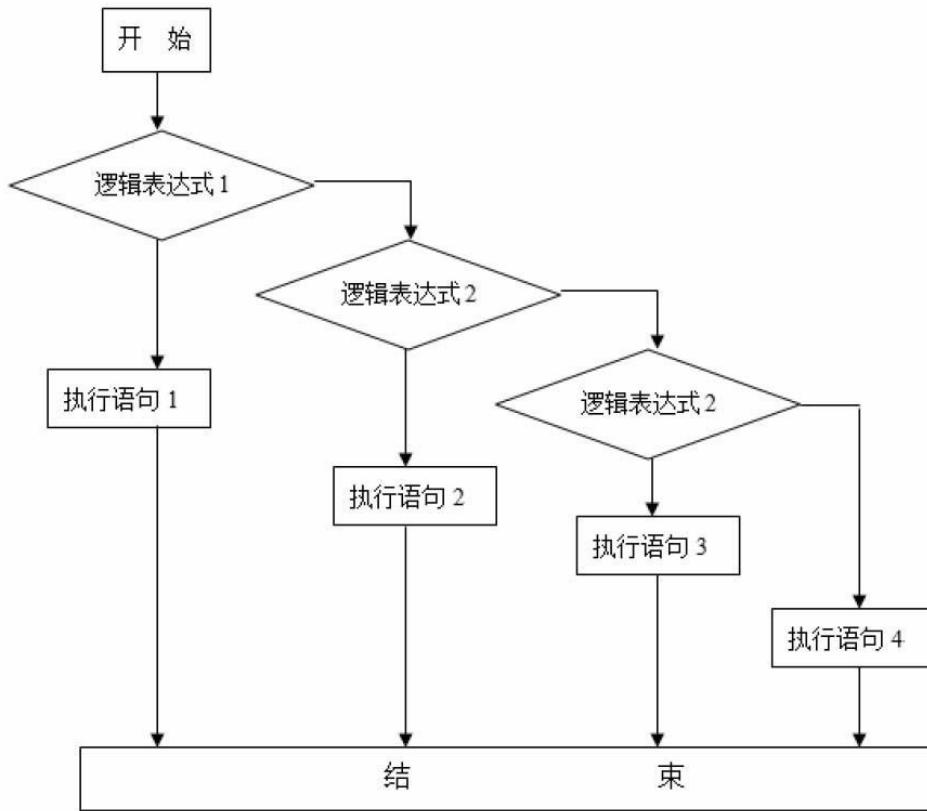


图14-1 嵌套if...else结构执行顺序示意图

嵌套if...else结构一般用于存在多种条件需要判断的情况，如对于判断两个数的关系来说，就可以使用该结构。其使用方式如示例脚本14-12.sh所示。

```

#示例脚本14-12.sh  if else结构的复杂使用
#!/bin/bash

echo if结构在gawk中的使用
gawk '{if ($1 > $2) {print $1 "大于" $2}
      else if($1 < $2) {print $1 "小于" $2}
      else {print $1 "等于" $2}}' data.txt
  
```

为示例脚本14-12.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ cat data.txt
10 20
10 5
10 10
ben@ben-laptop:~$ chmod u+x 14-12.sh
ben@ben-laptop:~$ ./14-12.sh
if结构在gawk中的使用
10小于20
10大于5
10等于10
```

通过示例脚本14-12.sh的执行结果可以看出，比较偏向于实际应用的条件结构是使用复杂的if....else结构，使用该结构可以对在实际中存在的各种情况进行逐一处理，而不会漏下任何的情形。

14.3.2 循环结构的使用

循环结构也可以用于gawk程序，用来对获取的数据进行循环处理。用于gawk结构的循环结构包括以下两种：

- **while**结构
- **for**结构

while结构的基本形式如下：

```
while(逻辑表达式)
{
语句;
}
```

while结构在执行时需要首先判断逻辑表达式的执行结果，如果逻辑结果为真，那么while结构中的语句将被执行。而如果逻辑表达式的执行结果为假，那么语句将不会被执行。while结构的基本使用如示例脚本14-13.sh所示。

```
#示例脚本14-13.sh  while结构在gawk中的使用
#!/bin/bash

echo while结构在gawk中的使用
gawk '{i=1; sum=0;while(i <= NF) {i++; print $2}}' data2.txt
```

为示例脚本14-13.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ cat data2.txt
1 11
2 22
3 33
4 44
ben@ben-laptop:~$ chmod u+x 14-13.sh
ben@ben-laptop:~$ ./14-13.sh
while结构在gawk中的使用
```

```
11  
11  
22  
22  
33  
33  
44  
44
```

通过示例脚本14-13.sh的执行结果可以看出，使用循环结构可以循环处理每一行文本记录，而操作方式可以按照实际的需要进行选择。

for结构的使用方式和while结构的使用方式类似，其基本格式如下：

```
for (表达式1; 表达式2; 表达式3)  
{  
    语句;  
}
```

在for结构中，需要使用3个表达式，其中表达式1一般使用来作为循环变量的赋值表达式，使得循环变量具有一个初始值。表达式2一般是一个逻辑表达式，用来判断循环是否可以执行。如果表达式2的运算结果为逻辑真，那么循环就可以正常执行，如果表达式2的执行结果为假，那么循环就不再被执行。而表达式3则是一个循环变量变化的方式，从而避免循环一直执行。for结构的一般使用方式如示例脚本14-14.sh所示。

```
#示例脚本14-14.sh  for结构在gawk中的使用
#!/bin/bash

echo  for结构在gawk中的使用
gawk '{
sum=0;
for(i = 1;i <= NF;i++)
{print $1}
}' data2.txt
```

为示例脚本14-14.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ cat data2.txt
1 11
2 22
3 33
4 44
ben@ben-laptop:~$ chmod u+x 14-14.sh
ben@ben-laptop:~$ ./14-14.sh
for结构在gawk中的使用
1
1
2
2
3
```

3

4

4

通过示例脚本14-14.sh的执行结果可以看出，在脚本中使用for循环结构可以实现命令的循环处理。执行的命令在每获取一行文本之后，都会得到执行。

14.3.3 循环结构控制语句

在使用循环结构时，如果符合循环执行的条件，循环结构将会无限期地执行下去。然后很多时候，当循环在满足某些条件时，就结束循环的执行，但是此时仍然满足循环继续执行的条件。此时需要使用下列语句来控制循环结构的运行状况：

- **break**
- **continue**
- **next**

break一般用来结束整个循环的最内层。continue用来跳过后面的语句，结束本次循环，从而开始下一次的循环。而next语句从输入文件中读取一行，然后从头开始执行gawk脚本。这些语句的使用方式如示例脚本14-15.sh所示：

```
#示例脚本14-15.sh 循环结构控制语句结构在gawk中的使用
```

```
#!/bin/bash
```

```
echo 循环控制语句在gawk中的使用
```

```
gawk '{  
for(i = 1; i <= NF; i++)  
{if($1 == 3)  
{continue;}  
if ($1 == 4){break;}  
print $1}  
}'  
data3.txt
```

为示例脚本14-15.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ cat data3.txt
```

```
1  
2  
3  
4  
5
```

```
ben@ben-laptop:~$ chmod u+x 14-15.sh
```

```
ben@ben-laptop:~$ ./14-15.sh
```

```
循环控制语句在gawk中的使用
```

```
1  
2  
3  
4  
5
```

通过示例脚本14-15.sh的执行结果可以看出，当获取的字段值为3时，直接跳转到该层循环的末尾，而不再执行该层循环中剩余的语句。而当获取的字段值为4时，直接跳出内层循环。因此在输出结果中，没有3和4这两个字段。

虽然break命令和continue命令的执行结果是一样的，但是二者的执行过程和执行原理却不相同。

14.4 函数的使用

在gawk程序中，不但可以使用变量以及各种程序结构，还可以使用函数来对文本等进行各种处理。在gawk中，不但存在许多内置函数，还可以自定义函数。本节将介绍在gawk中如何使用函数。

内置函数就是在gawk中可以不用定义而直接使用的函数，这部分函数类似于高级编程语言中的库函数，用户只需要按照函数的要求进行调用即可。在gawk中内置函数一般分为以下几类：

- 算术函数
- 字符串函数
- 时间函数
- 其他函数

在上面的函数中，每一部分都包含许多函数，本节将详细介绍这些函数的使用方式。

14.4.1 算术函数的使用

在gawk中，数学算术函数包括在数学中常用的算术操作，如开平方、求幂等。常用的算术函数如表14-3所示。

表14-3 常用的算术函数

函数名	作用
sin(x)	返回x的正弦值
cos(x)	返回x的余弦值
log(x)	返回x的自然对数
int(x)	对x进行取整操作
sqrt(x)	取x的平方根
exp(x)	返回x幂函数
atan2(y,x)	返回y/x的反正切
rand()	返回任意随机数
srand()	将rand()函数的种子值设置为x参数的值，如果省略expr参数将使用某天的时间，返回先前的种子值

通过表14-3中的算术函数可以实现一般的算术运算。在计算正弦、余弦时，使用的参数表示的是弧度，而不是普通的数值。而对于其他的函数来说，都能返回正常的数值。这些函数的使用方式如示例脚本14-16.sh所示。

```
#示例脚本14-16.sh 数学函数在gawk中的使用
```

```
#!/bin/bash

echo 使用取整函数
gawk '{num=int($1); print num }' data4.txt
echo

echo 使用取平方根函数
gawk '{num=sqrt($1);print num}' data4.txt
echo

echo 使用取对数函数
gawk '{num=log ($1);print num}' data4.txt
```

为示例脚本14-16.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$cat data.txt
10.123
30
10
ben@ben-laptop:~$ chmod u+x 14-16.sh
ben@ben-laptop:~$./14-16.sh
使用取整函数
10
30
10
```

使用取平方根函数

3.18167

5.47723

3.16228

使用取对数函数

2.31481

3.4012

2.30259

通过示例脚本14-16.sh的执行结果可以看出，在gawk中，也可以像在其他环境中使用数学函数那样进行各种数学计算。

14.4.2 字符串处理函数的使用

对于字符串的处理是gawk程序的主要任务之一。常用的字符串函数如表14-4所示。

表14-4 常用的字符串函数

函数	作用
index(String1, String2)	返回字符串String2在String1中的位置，如果出现多个位置，只返回第一个
length [(String)]	返回String的长度，如果未指定String，则返回整个记录的长度

length [(String)]	返回String的长度，如果未指定String，则返回整个记录的长度
substr(String, M, [N])	返回字符串中从位置M开始的后N个字符，如果不指定个数，将返回从位置M开始的所有字符
match(String, Ere)	返回String中Ere的位置，Ere参数指定一个扩展正则表达式
split(String, A, [Ere])	字符串分割
tolower(String)	将String中的大写字符转换成小写字符
toupper(String)	将String中的小写字符转换成大写字符
sprintf(Format, expr1, expr2, ...)	格式化生成字符串，使用expr代替格式中的占位符

字符串函数可以用来获取子字符串在字符串中的位置，并能够实现字符串的截取、分隔等操作，还能够将字符串进行大小写的转换。关于字符串函数的使用方式如示例脚本14-17.sh所示。

```
#示例脚本14-17.sh 字符串处理函数在gawk中的使用
#!/bin/bash

echo 获取字符h在第一个字段中的位置
gawk '{print index($1,"h") }' str.txt
echo

echo 获取长度
```

```
gawk '{print length($0) }' str.txt  
echo
```

echo将字符转换为大写字符

```
gawk '{print toupper($2)}' str.txt
```

为示例脚本14-17.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ cat str.txt  
this is test  
hello world  
123 456 786
```

```
ben@ben-laptop:~$ chmod u+x 14-17.sh  
ben@ben-laptop:~$ ./14-17.sh
```

获取字符h在第一个字段中的位置

```
2  
1  
0  
0
```

获取长度

```
12  
11  
11  
0
```

大小写转换函数

IS

WORLD

456

通过示例脚本14-17.sh的执行结果可以看出，在gawk中，对字符串的操作也可以使用各种函数来进行。

14.4.3 时间函数的使用

时间的处理是一般程序中难以处理的数据类型之一，因为时间有着特殊的格式，并且不容易表示。在gawk程序中，可以使用函数对时间进行基本的操作，常用的时间函数如表14-5所示。

表14-5 常用的时间函数

函数名	作用
mktime(YYYY MM DD HH MM SS[DST])	返回从1970年1月1日到确定时间的描述
strftime([format [, timestamp]])	将时间戳timestamp按照特定的格式转为 时间字符串
systime()	返回从1970年1月1日开始到当前时间 (不计闰年) 的整秒数

时间函数一般首先使用systime()函数来获取当前的时间，然后再使用mktime()函数按照特定的格式转换成时间。还可以将时间转换成字符

串。时间函数的使用方式如示例脚本14-18.sh所示。

```
#示例脚本14-18.sh 时间函数在gawk中的使用
#!/bin/bash

echo "使用systime()获取秒数"
echo | gawk '{print systime()}'
echo

echo "使用mktime ()获取秒数"
echo | gawk '{print mktime ("2014 03 28 14 12 12")}'
```

为示例脚本14-18.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-18.sh
ben@ben-laptop:~$ ./14-18.sh
使用systime()获取秒数
2299349202

使用mktime ()获取秒数
2014 03 28 14 12 12
```

对于strftime()函数来说，可以使用特定的格式来指定时间信息，常用的格式如表14-6所示。

表14-6 strftime()函数的格式说明

格式符	说明
%D	表示日期，使用0填补空位
%e	表示日期，使用空格填补空位
%H	24小时格式的小时，用十进制表示
%I	12小时格式的小时，用十进制表示
%j	表示一年中的第几天，从1月1日开始算起
%m	月份，使用十进制表示
%M	分钟，使用十进制表示
%p	使用12小时表示法，使用AM或PM来区分上午还是下午
%S	秒，用十进制表示
%U	十进制表示的一年中的第几个星期，将星期天作为一个星期的开始
%w	十进制表示的星期几，将星期天表示为0
%W	十进制表示的一年中的第几个星期，将星期一作为一个星期的开始
%x	重新设置本地日期，格式为08/20/99
%X	重新设置本地时间，格式为00: 00: 00
%y	表示使用两位数字表示的年，如99表示1999年
%Y	表示当前月份

%Z	表示当前所在的时区（PDT）
%%	表示百分号（%）

在表14-6中可以看出，`strftime()`函数使用的日期格式和C语言中函数`strftime()`的格式相同，都能使用简单的字符表示日期，该函数的使用方式如示例脚本14-19.sh所示。

```
#示例脚本14-19.sh  strftime函数的使用
#!/bin/bash

echo 使用函数strftime输出到当前时间
echo | gawk '{print strftime("%D", systime())}'
echo | gawk '{print strftime("%e", systime())}'
echo | gawk '{print strftime("%w", systime())}'
```

为示例脚本14-19.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 14-19.sh
ben@ben-laptop:~$ ./14-19.sh
08/14/14
14
4
```

通过示例脚本14-19.sh的执行结果可以看出，使用函数`strftime()`可以将时间按照特定的格式输出，而不只显示秒数。这种操作更贴近于实际的需要。

14.5 小结

本章介绍了gawk程序在Shell编程中的使用方式，gawk编辑器类似于一个编程环境的工具，允许修改和重新组织文件中的数据。gawk程序和sed编辑器一样，都是从输入文件中以行为单位获取输入数据，然后使用既定的处理方式进行处理。

在gawk程序中，可以使用选项来决定从标准输入中获取输入数据，还是从特定的文件中获取数据，还可以使用其他选项来输出其他的結果。

在gawk中可以使用变量来实现数据的操作。gawk程序可以使用内置变量来确定gawk程序中的一些基本信息，还可以使用自定义变量来对数据进行各种操作。

在gawk中，还可以使用各种结构形式来对指定文件中的信息进行各种处理。可以使用条件结构来使得当满足某项条件时执行一些语句，可以使用if...else结构来实现。而循环结构一般用来循环执行某些语句，而在满足特定的条件下才会退出循环，一般可以使用while循环结构、for循环结构来实现对循环的处理。在使用循环结构时，还可以使用continue语句、break语句来实现对循环结构执行顺序的调整与控制。

函数也同样可以用在gawk程序中，其中使用频率最高的是一些内置函数，通过这些内置函数能够进行数学运算、字符串的常规处理以及对时间的各种操作。

第15章 脚本控制

脚本是一组命令的集合，在执行脚本时，由系统的脚本解释器程序也就是本书介绍的bash，将其中的命令翻译成机器可以识别的指令，并按程序顺序执行。因此在必要的时候，也需要对脚本的执行流程进行控制。本章将介绍几种控制脚本运行的方式。

本章的主要内容如下：

- Linux信号控制机制
- 开机运行脚本的方法
- 后台运行脚本的方法
- 脚本运行优先级管理

15.1 Linux信号控制

Linux信号实际是在软件层面上对中断机制的模拟，所以也称为中断。中断是Linux系统中一种进程间通信的方式。由于脚本的执行由操作系统启动解释器进程，解释器翻译脚本内编写的命令执行程序，因此脚本执行也以进程的形式在系统中运行。那么脚本就能够通过中断的方式进行脚本之间或者脚本与其他进程之间的通信。

15.1.1 Linux信号机制简介

信号可以从操作系统内核发往一个进程，也可以由一个进程发往另

外一个进程。信号主要用来通知进程发生了什么事件，不能给进程传递任何数据。信号的传递是异步的，也就是说一个进程在做任何事情的时候都有可能收到信号，因此进程对信号的处理对于进程的执行来说优先级是最高的，进程需要调用相应的处理过程对收到的信号做出响应，信号处理结束后进程再回过头来继续往下执行。

信号按照来源分可分为两种，一种是硬件来源（如按下键盘），另一种是软件来源（如最常用的用来发送信号的系统调用kill等函数）。

信号还可按照可靠性来分，一种是不可靠信号，由于早期Unix系统中的信号机制比较简单，后来在使用中暴露出很多问题，比如在某种情况下将导致对信号的错误处理，有的信号可能丢失，进程可能会忽略对一些信号的处理，或者重复处理某些信号。因此把早期的信号叫做不可靠信号；随着计算机技术的发展，后来人们发现有必要对原始的信号机制进行改进和补充。在后续的Unix操作系统中，信号的发送、响应和处理等功能得到了很大的改进，这些信号支持队列，不会丢失，不会重复处理。

注意

进程对各种信号有不同的处理方法。可以由进程指定处理函数，当收到信号时，由相应的处理函数来处理，或者是忽略某信号，对该信号不做任何处理。或者使用系统默认方式处理某信号。

Linux系统中的信号有很多，可以将这些信号归类为以下几种：

- 与进程终止执行相关的信号。当进程或者子进程终止时发出该

类信号。

- 与进程异常事件相关的信号。当发生如进程内存越界，或者企图写一个只读的内存区域的时候发出该类信号。
- 与在执行系统调用期间遇到不可恢复情况相关的信号。当发生比如在执行系统调用exec时，原有的资源还没有释放而目前系统资源又已经耗尽。
- 与执行系统调用时遇到非预测错误条件相关的信号。当执行一个并不存在的系统调用时。
- 在用户态下的进程发出的信号。当用户进程调用系统调用kill向其他进程发出信号时。
- 与终端交互相相关的信号。当用户关闭一个终端，或者按下break键等情况。
- 跟踪进程执行的信号。

通过kill -l命令可以列出系统支持的所有信号。Linux系统共有62个信号，其中编号1~31是非实时信号，编号34~64是实时信号。实时信号和非实时信号的区别在于当进程接到多个实时信号时，会将每个信号排队处理，而如果接到多个非实时信号，进程会将这几个信号合并为一个处理。Linux支持的非实时信号列表如表15-1所示。

表15-1 Linux支持的非实时信号及其意义

编号	信号	说明
1	SIGHUP	终止进程，挂断终端线路。前台进程都会附在某个终端中执行，该信号用来使进程与终端脱离
2	SIGINT	键盘中断

3	SIGQUIT	从键盘退出
4	SIGILL	非法的指令情况出现该信号
5	SIGTRAP	跟踪程序或者在程序中设置断点
6	SIGABRT	程序运行中发现问题并调用abort时产生
7	SIGBUS	总线错误，比如故障的内存访问
8	SIGFPE	浮点异常
9	SIGKILL	杀死进程
10	SIGUSR1	用户自定义信号1
11	SIGSEGV	无效的内存引用
12	SIGUSR2	用户自定义信号2
13	SIGPIPE	管道故障，向一个没有读进程的管道写数据
14	SIGALRM	计时器到时
15	SIGTERM	终止进程，进程自行结束
17	SIGCHLD	当子进程停止或者退出的时候通知父进程
18	SIGCONT	继续执行一个停止的进程
19	SIGSTOP	不在终端中停止进程
20	SIGTSTP	在终端中停止进程

21	SIGTTIN	后台进程从终端读取
22	SIGTTOU	后台进程向终端写入
23	SIGURG	套接字紧急信号
24	SIGXCPU	超过CPU时间限制
25	SIGXFSZ	超过文件大小限制
26	SIGVTALRM	该进程占用的CPU时间
27	SIGPROF	审查计时器超时
28	SIGWINCH	改变窗口大小
29	SIGIO	I/O准备就绪
30	SIGPWR	加电失败
31	SIGSYS	非法的系统调用

15.1.2 使用**Shell**脚本操作信号

Linux操作系统可以通过键盘发送信号，可以通过kill命令来发送信号，也可以通过编程的方式（如调用系统API）发送信号。本章着重介绍通过键盘和命令的方式发送信号。

通过键盘发送信号，就是在进程运行的过程中通过键盘按键向进程发出信号，当然如果使用此方式，接收信号的进程必须在终端前台运行，因为只有在终端前台才能捕获到键盘按键，如示例脚本15-1.sh所

示。

```
#示例脚本15-1.sh 使用键盘按键向脚本发送信号
#!/bin/bash

while ((1))
do
echo "hello shell script"
sleep 1
done
```

为示例脚本15-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 15-1.sh
ben@ben-laptop:~$ ./ 15-1.sh
hello shell script
hello shell script
hello shell script
hello shell script
^C
ben@ben-laptop:~$
```

通过示例脚本15-1.sh的执行结果可以看出，脚本运行后，在屏幕上不断地输出“hello shell script”，当按下键盘上的Ctrl + C键后，屏幕停止输出字符串，脚本停止运行，这相当于给该进程发送了一个SIGINT信号，进程收到SIGINT信号后中断执行。

在Linux终端下还可以使用kill命令来产生并发送信号。kill命令能够将指定的信号发送给指定的进程或者进程组。如果没有指定信号，它将以SIGTERM作为默认的信号发送。kill命令通常与ps命令配合使用来完成向进程发送信号的任务。使用ps命令能列出进程的pid，kill命令就可以通过这个pid向其发送信号，如示例脚本15-2.sh所示。

```
#示例脚本15-2.sh 使用键盘按键向脚本发送信号
#!/bin/bash

echo 查看进程是否存在
ps -ef | grep 15-1.sh |grep -v grep

echo 杀掉进程15-1.sh
ps -ef | grep 15-1.sh | awk '{print $4}' | xargs kill -9

echo 查看进程是否存在
ps -ef | grep 15-1.sh
```

为示例脚本15-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 15-2sh
ben@ben-laptop:~$ ./ 15-2.sh
查看进程是否存在
ben          2990  2880  0 00:00 pts/2      00:00:01 /bin/bash ./
杀掉进程15-1.sh
查看进程是否存在
```

通过示例脚本15-2.sh的执行结果可以看出，使用kill命令能够直接将进程停掉。当进程被停止之后，使用ps命令不再显示出来。

注意

kill命令前面的内容最终是为了获取进程号，对这部分内容可以参照其他的章节进行学习。

在Linux系统中，还可以使用trap命令来捕捉获取信号，然后设定进行怎样的特定处理。trap命令的常用方式如表15-2所示。

表15-2 常见的trap捕获命令方式

命令	说明
trap "cmds" signals	当Shell接收到signals指定的信号时，执行cmds命令
trap signals	如果没有指定命令部分，系统将恢复默认的处理方式
trap "" signals	忽略列出的信号
trap -p signal	打印当前signal的trap设置

接下来介绍一个具体的捕获信号的例子。首先确定要捕获的信号，当然用户可以捕获任何一个系统中存在的信号，因为Shell本身需要捕捉这个信号进行内存转储。在此例中为了不影响系统正常工作，将使用系统预留给用户自己定义的信号SIGUSR1。接下来在终端中输入以下语

句：

```
ben@ben-laptop:~$trap "echo hello" USR1
```

当捕获到USR1信号时，系统将执行第一个参数内指定的命令echo hello，从而在终端中输出hello字符串，否则会报出command not found的错误。然后使用kill命令发送信号给系统进程，当系统进程捕获到信号时会触发相应的处理方法，如下面的操作所示：

```
ben@ben-laptop:~$kill -s USR1 0
```

在上面的操作中，0号进程是Linux系统的init进程，使用kill命令发送信号给该进程，当它收到信号后会做相应的处理，即执行命令echo hello，然后就会在屏幕上会看到hello字样。

注意

trap命令不能捕捉信号11（内存引用错误）的错误，并且trap命令捕捉到信号后执行的命令必须是一个系统能够找到的命令。

15.2 进程的控制

Unix/Linux系统是一个多任务操作系统，它的多任务管理功能是非常强大的。接下来的内容将介绍后台任务。所谓后台任务就是当任务运行的时候不需要用户干预，或者是用户通过一些其他的方式（比如向进

程发送信号)才能干预到的任务。后台任务的运行不会干扰到用户做其他事情，它们会悄悄地在后台执行，当然，一旦任务中出现异常它将把异常信息写入日志，这样用户可以查看到任务的运行情况。通过以下章节，读者朋友能够了解运行后台任务的几种方式。

15.2.1 后台运行符介绍

在Linux系统中，在后台执行只需要在启动脚本时使用后台运行符，后台运行符使用符号“&”表示。在使用时，可以将后台运行符放到执行的命令的最后面，那么，程序在执行以后，就会自动的转到后台，如示例脚本15-3.sh所示。

```
#示例脚本15-3.sh 后台运行符的使用
#!/bin/bash

while ((1))
do
sleep 1
done
```

为示例脚本15-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 15-3.sh
ben@ben-laptop:~$ ./ 15-3.sh
^C
```

```
[1]+  Terminated                  ./15-3.sh
ben@ben-laptop:~$
```

对于示例脚本15-3.sh脚本来说，脚本执行以后，终端一直处于执行脚本的状态中，这时用户什么事情也做不了，只能等待程序一直不停地执行下去，除非用户在键盘按下Ctrl+C键才能够终止进程。

如果用户在执行完程序后需要做一些其他的交互，就可以通过在命令后面增加“&”符号的方式来执行，如下面的执行方式所示：

```
ben@ben-laptop:~$ ./15-3.sh &
[1] 4871
ben@ ben-laptop:~$
```

当使用该种方式执行了脚本15-3.sh以后，终端将不再等待脚本执行结束，而是直接显示新的提示符，这样就可以执行新的命令，而不需要等待脚本运行完毕。

使用后台运行符执行脚本以后，屏幕上会显示一个方括号，里面有个数字代表当前作业的编号，这个作业编号是系统自动分配的，从1开始。通过jobs命令可以查看作业列表。在作业编号的后面跟着一串数字，表示该进程ID。通过ps命令可以查看该进程的详细情况，如下所示。

```
ben@ben-laptop:~$ ./15-3.sh &
[1] 4871
ben@ ben-laptop:~$ ps -ef | grep 4871 | grep -v grep
```

```
ben           4871  2880  0  01:04 pts/2    00:00:00 /bin/bash ./
ben@ben-laptop:~$ jobs
[1]+  Running                  ./15-1.sh &
ben@ben-laptop:~$ jobs -l
[1]+  5382 Running              ./15-1.sh &
```

使用后台运行符可以将一些平时不需要在前台查看而直接运行的进程放在后台执行，这样就可以既不影响前台的正常使用，也不影响后台进程的运行。

15.2.2 运行进程的控制

在Linux系统中，进程可以在前台运行，也就是在命令行中直接运行，需要等待进程结束后才可以进行其他的操作。某些进程还可以在后台运行，此时不需要进行任何的等待而直接进行其他的操作。

对于这两种类型的进程，可以通过命令或其他的操作进行控制。常用的操作方式包括以下3种：

- 使用**jobs**命令；
- 通过**fg**和**bg**命令；
- 通过键盘键入命令。

jobs命令的主要作用是显示当前环境中的所有的任务，而使用不同的参数可以显示不同的任务信息，常用的选项如表15-3所示。

表15-3 **jobs**命令常用选项

选项	作用
-l	列出进程ID
-n	只列出那些自从上次用户修改状态后的进程信息
-r	只列出状态是running 的任务
-s	只列出状态是stopped 的任务

表15-3只列出了jobs命令的常用选项，这些选项的使用方式如示例脚本15-4.sh所示。

```
#示例脚本15-4.sh  jobs命令的使用
#!/bin/bash

echo 使用-l选项
jobs -l
echo

echo 显示状态是running的任务
jobs -r
```

为示例脚本15-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 15-4.sh
ben@ben-laptop:~$ ./ 15-4.sh
使用-l选项
```

```
[1]+  5382 Running
```

```
./15-1.sh &
```

显示状态是running的任务

在后台运行的命令可以使用jobs命令查看，通过使用fg命令和bg命令也可以干预进程的执行。fg命令的作用是使后台中的命令调至前台继续运行，而bg命令一般和Ctrl+Z配合使用，Ctrl+Z用来暂停前台程序的运行，而bg命令则是继续执行暂停的命令。这3个命令的使用方式如下所示。

```
ben@ben-laptop:~$ ./15-1.sh
hello shell script
hello shell script
hello shell script
[1] + Stopped (SIGTSTP)    ./15-1.sh
ben@ben-laptop:~$ bg
[1]      ./ 15-1.sh&
ben@ben-laptop:~$ ps -ef | grep 15-1| grep -v grep
ben  8913158 36503620    1 10:02:22  pts/5    0:00 /bin/bash ./
```

使用了Ctrl+Z命令暂停程序运行之后，可以使用bg命令恢复程序的运行，而此时程序自动转为后台运行，而不在前台运行了。

15.2.3 nohup命令的使用

对于后台运行的命令来说，一般使用后台运行符“&”来表示。当使用后台运行符时，程序只在该终端的后台运行，而没有真正实现后台运行。如果终端关闭，那么程序也会随之被关闭。为了真正实现程序的后台运行，则需要使用nohup命令。

nohup命令的使用方式如下：

```
nohup 命令 &
```

在使用nohup命令时，需要在执行的命令前面加上nohup命令，然后在执行命令的后面添加后台运行符“&”，当程序运行以后，就可以真正地实现程序的后台运行了。即使当前的终端关闭，程序也会在进程列表中存在。

在使用了nohup命令以后，本来需要输出到屏幕上的标准输出将直接输出到文件名为nohup.out文件中，该文件是执行nohup命令以后创建的日志文件，文件中存储了标准输出的内容，对于一般的操作方式来说，如果多个进程同时使用nohup命令来启动，那么所有的日志文件都会存储在nohup.out文件中，此时就需要使用重定向来将程序的输出进行重定向，从而对不同的进程的输出结果进行区分。nohup命令的使用方式如示例脚本15-5.sh所示。

```
#示例脚本15-5.sh nohup命令的使用
```

```
#!/bin/bash
```

```
echo 使用nohup命令执行15-3.sh
```

```
nohup ./15-3.sh &
```

```
echo  
echo 使用后台运行符执行15-1.sh  
. ./15-1.sh &  
echo  
echo 查看进程  
ps -ef | grep 15-* | grep -v grep
```

为示例脚本15-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 15-5.sh
ben@ben-laptop:~$ ./ 15-5.sh
使用nohup命令执行
[2] 2982
nohup: ignoring input and appending output to `nohup.out'

使用后台运行符执行
[2] 2839

查看进程
ben          2827  2753  0 23:37 pts/0    00:00:00 /bin/bash
ben          2982  2753  0 23:39 pts/0    00:00:00 /bin/bash
```

当关闭中断后，打开一个新的终端，再次查看进程，其操作方式如下：

```
ps -ef | grep 15-* | grep -v grep  
ben          2982      1  0 23:39 ?          00:00:00 /bin/bash ./
```

通过示例脚本15-5.sh的执行结果可以看出，当使用了nohup命令时，程序真正实现了后台执行。而使用后台运行符后执行的命令，当终端关闭后，进程随之消失。

15.3 脚本运行的优先级

在Linux系统中，脚本、程序、命令等可能同一时刻都需要运行，而CPU在一个时刻只能运行其中的一个，那么到底先运行哪个，后运行哪个呢？这需要有一个明确的标识，否则，CPU将不知道该首先执行哪一个，然后执行哪一个。这个标识就是优先级。本节就重点介绍优先级的相关知识。

15.3.1 优先级介绍

在Linux系统中，脚本、程序、命令的执行都按照进程的方式执行，进程执行的先后顺序是按照进程的优先级来确定的，进程优先级是在进程创建时，储存在进程控制块（Process Control Block，PCB）中的，当CPU有空间时，就会遵循进程调度的方式，按照优先级的高低来执行进程，直到进程都执行完毕。

在系统中，进程的优先级一般包括实时优先级和静态优先级。在任何时刻，实时进程的优先级都要高于普通进程，实时进程之间将按照进

程的优先级来确定首先处理哪个进程，在Linux 2.6内核中，实时进程的优先级范围为0~99，而普通进程的优先级范围是100~139。因此，数值越小，进程的优先级就越高。

在Linux系统中，可以通过ps命令来获取进程的优先级，首先使用ps -l命令显示当前系统中的进程，如下所示：

```
ben@ben-laptop:~ #ps -l
  F S UID   PID   PPID    C PRI  NI ADDR SZ WCHAN
200001 A 245 15532154 6816142  0 60   20 4174c3480 97
240001 A 245 6816142 6750270  1 60   20 7680480 82
```

ps命令能够显示当前系统中所有正在运行的进程，并能够显示这些进程的详细信息。其中，字段PRI和NI是和进程的优先级相关的字段。字段PRI是进程的优先级，这个值越小，进程的优先级别越高。如果进程是实时进程，那么当前的优先级就表示为静态优先级，如果进程是非实时进程，那么当前的优先级就表示为动态优先级。PRI默认的取值范围是0~MAX_RT_PRIO，而MAX_RT_PRIO在Linux系统中一般配置为100，因此，进程优先级的取值范围是从0到99。

注意

进程的优先级问题涉及操作系统中进程调度的相关知识，本节只是简单地讲述一下优先级相关的内容，如果读者想更进一步了解其中的知识，可以参照相关资料进行学习。

15.3.2 使用nice指定优先级

在使用ps-l命令显示当前系统中运行的进程的详细信息中，nice值是进程优先级的修正值， nice值和优先级的关系如下：

$$\text{PRI值} = \text{PRI值(旧值)} + \text{NICE值}$$

上面的表达式表示了NICE值和进程优先级的关系。进程最终的优先级是PRI值和NICE值的和。NICE的值可以是正值，也可以是负值。当NICE为正值时，会使得进程的优先级降低，而当NICE为负值时，则可以提高进程的优先级。

nice值的范围是-20~19，而相对应的进程的静态优先级范围在100~139之间。nice数值越大就使得static_prio越大，最终进程优先级就越低。

设置进程的nice值使用nice命令， nice命令的使用方式如下：

```
 nice [-n <优先等级>] [--help] [--version] [执行指令]
```

nice命令可以用来改变程序执行的优先权等级。通过影响进程的PRI值来间接地改变进程的优先级。常用的选项如表15-4所示。

表15-4 nice命令的常用选项

选项	作用
-n	设置执行命令的优先级

--help	显示帮助
--version	显示nice命令的帮助信息

在表15-4中，--help选项和--version选项的使用方式和其他命令的使用方式一样，用来显示nice命令的一些帮助信息和基本信息。选项-n主要用来设置命令的优先级，范围是-20~19，数值越小，设置成功后进程的优先级越高。nice命令的使用方式如示例脚本15-6.sh所示。

```
#示例脚本15-6.sh nice命令的使用
#!/bin/bash

echo "设置进程的优先级"
nice -10 ./nice-test.sh &
echo

echo 查看进程信息
ps -efl | grep 15-1.sh | grep -v grep
```

为示例脚本15-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 15-6.sh
ben@ben-laptop:~$ ./ 15-6.sh
设置进程的优先级

查看进程信息
  0 S ben          6009  5631  0   90  10 -  1293 wait    00:00 pts.
```

通过示例脚本15-6.sh的执行结果可以看出，使用nice命令可以直接对进程执行的优先级产生影响，该影响在进行进程调度时，将会发挥重大的作用。

注意

nice值不是进程的优先级，只能对进程的优先级产生影响，并且只有root用户才能使用nice命令设置进程的优先级。

15.3.3 使用renice重置优先级

当进程的优先级设定以后，还可以使用renice命令重新设置。renice命令的使用方式如下：

```
renice [优先等级][-g <程序群组名称>...] [-p <程序识别码>...] [-u <用
```

renice命令可以通过选项来设置群组、程序的优先级，设定的范围也是从-20到19。而且能够使用renice命令的只有系统管理用户，普通用户无法改变进程的优先级。renice命令的使用方式如示例脚本15-7.sh所示。

```
#示例脚本15-7.sh renice命令的使用
#!/bin/bash
```

```
echo 查看进程信息
```

```
ps -efl | grep 15-1.sh | grep -v grep  
echo
```

```
echo 使用renice命令重置进程的优先级
```

```
renice 16 -p 6009  
echo
```

```
echo 查看进程信息
```

```
ps -efl | grep 15-1.sh | grep -v grep
```

为示例脚本15-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 15-7.sh
```

```
ben@ben-laptop:~$ ./ 15-7.sh
```

查看进程信息

```
0 S ben      6009  5631  0  90  10 -  1293 wait    00:00 pts.
```

使用renice命令重置进程的优先级

```
0 S ben      6009  5631  0  90  16 -  1293 wait    00:00 pts.
```

通过示例脚本15-7.sh的执行结果可以看出，使用renice命令可以对进程执行的优先级进行进一步的更改，而进程的优先级也可以通过使用renice命令做出改变，从而使优先级不会一成不变。

注意

`renice`命令也是只能通过影响进程的`nice`值来改变进程的优先级，而不是直接能够改变进程的优先级。

15.4 小结

本章主要介绍了在Linux系统中如何对脚本执行流程进行控制。可以通过Linux系统的信号机制实现对脚本的控制。在Linux系统中，通过键盘的输入、命令都可以向脚本发送信号。通过键盘可以停止或暂停脚本的运行，而使用`kill`命令等也可以实现终止或暂停脚本的运行。

开机运行脚本主要是借助于Linux系统在开机加载系统的过程中，需要执行`.profile`中的语句，如果需要在开机时就运行某些脚本，那么就将要执行的脚本放到该文件的后面，这样，开机后该脚本也就已经运行了。

脚本除了可以设定为在开机时运行，还可以设定为后台运行。当脚本设定为后台运行以后，就可以始终在运行，而不会干扰到其他进程的运行，但是，对于后台运行的脚本来说，输出的内容应该重定向到其他的文件中，而不是使用默认的标准输出。否则，输出的内容还会显示在计算机的屏幕上。

进程在执行时按照优先级来决定哪个脚本首先被执行，可以使用`nice`来设置进程的`nice`值从而进一步影响进程的优先级。当使用`nice`命令设置了进程的优先级以后，还可以使用`renice`对进程的`nice`值进行重置。

第16章 脚本运行的优化

前面的章节中讲述了Shell脚本的编写和使用方式，这些脚本都是简单的文字性描述，运行结果也都是文字性的表达，缺乏图形化的运行效果。在实际运用中，运行过程会显得非常不美观。为了达到美观的效果，运行Shell脚本时，可以对其进行一定程度的优化，使得Shell脚本的运行过程脱离纯文字。

本章的主要内容如下：

- 窗口的添加，即dialog软件包的使用
- 在Shell脚本运行时添加颜色效果
- 菜单的创建

16.1 添加窗口

现在最常用的是Windows系统和Linux系统，在使用这两个系统的过程中，经常会出现各种对话框的情形，而在Shell脚本的运行过程中，也可以使用dialog软件包来实现窗口的效果。本节就讲述如何在Shell脚本中使用窗口。

16.1.1 dialog软件的安装

dialog软件包是Linux系统中常用的软件包，该软件包最初由Savio Lam设计，它利用ANSI转义控制码在文本环境中创建标准的Windows对

话框，从而可在Shell脚本中使用对话框，实现文本的交互。

如果用户是第一次使用dialog软件，需要安装dialog的相关内容。因为对于系统来说，默认情况下不安装dialog软件。可以选择使用apt-get命令进行安装，安装过程如下：

```
ben@ben-laptop: sudo apt-get install dialog
[sudo] password for ben:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列【新】软件包将被安装:
  dialog
升级了 0 个软件包, 新安装了 1 个软件包, 要卸载 0 个软件包, 有 313 个软
需要下载 271KB 的软件包。
解压缩后会消耗掉 1,466KB 的额外空间。
获取: 1 http://cn.archive.ubuntu.com/ubuntu/ lucid/universe d...
下载 271KB, 耗时 1秒 (158KB/s)
选中了曾被取消选择的软件包 dialog。
(正在读取数据库 ... 系统当前总共安装有 162467 个文件和目录。)
正在解压缩 dialog (从 .../dialog_1.1-20080819-1_i386.deb) ...
正在处理用于 man-db 的触发器...
正在设置 dialog (1.1-20080819-1) ...
```

运行apt-get命令之后，系统首先查找dialog程序相关的软件包以及软件包之间的依赖关系，当找到之后，就会按照软件包的依赖关系进行自动安装。当安装完成后，命令运行结束，dialog就能正常使用了。

在dialog的安装过程中，要保证网络畅通，并且只有root用户才能进行软件的安装。如果以普通用户身份进行安装，则需要使用sudo命令来提升命令的执行权限。

16.1.2 dialog命令的帮助选项

dialog命令在创建不同的窗口时，需要使用不同的输出格式，面对数量众多的选项和参数，一般人不会选择将所有的内容都记住，这样会浪费时间，而且在稍长时间不用后，还会遗忘。幸好dialog命令提供了一个--help选项，能够详细展示dialog命令的使用信息，如下面的内容所示：

```
ben@ben-laptop:~$ dialog --help
cdialog (ComeOn Dialog!) version 1.1-20080819
Copyright 2000-2007, 2008 Thomas E. Dickey
This is free software; see the source for copying conditions
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

* Display dialog boxes from shell scripts *

Usage: dialog <options> { --and-widget <options> }
where options are "common" options, followed by "box" options

Special options:
[--create-rc "file"]

Common options:
```

```
[--ascii-lines] [--aspect <ratio>] [--backtitle <backtitle>]
[--begin <y> <x>] [--cancel-label <str>] [--clear] [--color <str>]
[--column-separator <str>] [--cr-wrap] [--default-item <str>]
[--defaultno] [--exit-label <str>] [--extra-button]
[--extra-label <str>] [--help-button] [--help-label <str>]
[--help-status] [--ignore] [--input-fd <fd>] [--insecure]
[--item-help] [--keep-tite] [--keep-window] [--max-input <n>]
[--no-cancel] [--no-collapse] [--no-kill] [--no-label <str>]
[--no-lines] [--no-ok] [--no-shadow] [--nook] [--ok-label <str>]
[--output-fd <fd>] [--output-separator <str>] [--print-max <n>]
[--print-size] [--print-version] [--quoted] [--separate-out]
[--separate-widget <str>] [--shadow] [--single-quoted] [--size <n>]
[--sleep <secs>] [--stderr] [--stdout] [--tab-correct] [--text <str>]
[--timeout <secs>] [--title <title>] [--trace <file>] [--tcl]
[--version] [--visit-items] [--yes-label <str>]
```

Box options:

--calendar	<text> <height> <width> <day> <month> <year>
--checkbox	<text> <height> <width> <list height> <tag1> <tag2>
--dselect	<directory> <height> <width>
--editbox	<file> <height> <width>
--form	<text> <height> <width> <form height> <label height>
--fselect	<filepath> <height> <width>
--gauge	<text> <height> <width> [<percent>]
--infobox	<text> <height> <width>
--inputbox	<text> <height> <width> [<init>]
--inputmenu	<text> <height> <width> <menu height> <tag1> <tag2>
--menu	<text> <height> <width> <menu height> <tag1> <tag2>

```
--mixedform      <text> <height> <width> <form height> <labe
--mixedgauge     <text> <height> <width> <percent> <tag1> <i
--msgbox         <text> <height> <width>
--passwordbox    <text> <height> <width> [<init>]
--passwordform   <text> <height> <width> <form height> <labe
--pause          <text> <height> <width> <seconds>
--progressbox    <height> <width>
--radiolist      <text> <height> <width> <list height> <tag1>
--tailbox        <file> <height> <width>
--tailboxbg      <file> <height> <width>
--textbox         <file> <height> <width>
--timebox        <text> <height> <width> <hour> <minute> <se
--yesno          <text> <height> <width>
```

```
Auto-size with height and width = 0. Maximize with height and width = -1.
Global-auto-size if also menu_height/list_height = 0.
```

通过上面的运行结果可以看出，使用`--help`可以将`dialog`命令中所有的窗口类型以及创建这些窗口时能够使用的参数显示出来，这样就免去了记忆所有内容的麻烦。只需要记住：遇到问题时先使用`--help`选项即可。

16.1.3 dialog命令的使用

`dialog`软件包是由`dialog`命令来实现的，`dialog`命令的一般使用方式

如下：

```
dialog --窗口类型 --参数
```

dialog能够将需要显示的内容显示在确定的窗口上。窗口类型前面一般使用双横线“--”进行标识。dialog命令常用的窗口类型如表16-1所示。

表16-1 dialog命令使用的命令行参数

命令行参数	作用
calendar	显示日历
checklist	复选框
form	表单
editbox	编辑框
fselect	文件选框，即平时上传本地文件时的工具
gauge	进度条
infobox	弹出一个文本信息，不需要等待回应
inputbox	文本框
inputmenu	可编辑菜单
menu	选择菜单
msgbox	弹出一个文本信息，需要用户选择，并单击OK

pause	暂停页
passwordbox	密码文本框，输入的任何内容显示为*
passwordform	密码表单
radiolist	单选按钮选框
tailbox	显示tail命令类似内容
tailboxb	加上背景的tailbox
textbox	文本窗口，显示内容，如果一屏显示不了，可以滚动地在整个窗口显示
timebox	选择时间
yesno	显示yes和no按钮

通过表16-1中描述的dialog支持的窗口可以看出，在Shell脚本中，通过dialog命令的特定选项就可以创建Windows系统中常用的窗口类型，如示例脚本16-1.sh所示。

```
#示例脚本16-1.sh dialog命令创建简单的Windows窗口
#!/bin/bash

echo 创建普通的消息窗口
dialog --msgbox "hello world" 10 20
```

为示例脚本16-1.sh赋予可执行权限后执行脚本，结果如图16-1所示。

```
ben@ben-laptop:~$chmod u+x 16-1.sh  
. / 16-1.sh
```



图16-1 脚本16-1.sh执行结果

通过示例脚本16-1.sh的执行结果可以看出，使用--msgbox命令创建了一个尺寸为10×20的小窗口，这个窗口类似于Windows系统中的消息窗口，在添加特定的信息之后，就能实现不同的功能。

注意

在创建窗口的脚本运行结果中，为了使得结果更加直观，可使用截图的方式来表示结果。

dialog命令还可以使用参数来丰富创建的窗口的内容，如添加标题、背景颜色、显示某些按钮等。dialog命令常用的选项如表16-2所示。

表16-2 dialog命令常用选项

选项	作用
--backtitle	背景标题
--begin y x	用于指定窗口的位置, y为离上边缘的距离, x为离左边缘的距离
--colors	使用\Z开始, \ZN结束框起来
--defaultno	确定默认按钮是【是】按钮还是【否】按钮
--default-item	如果是个多选框, 默认选中的条目
--insecure	使用密码框时是否显示密码为*, 默认显示为星号
--nocancel	不显示【取消】按钮
--nook	不显示【确认】按钮
--no-shadow	文本框是否有阴影, 默认有阴影
--ok-label	默认label显示【确认】按钮
--timeout	如果给出一个对话框, 什么都不做的话, 超时时间是多少
--title	确定标题内容

在表16-2展示了dialog命令常用选项, 可以选择一个选项单独使用, 还可以多个选项共同使用。关于这些选项的简单用法如示例脚本16-2.sh所示。

```
#示例脚本16-2.sh dialog命令常用选项的使用
```

```
#!/bin/bash

echo 创建有背景标题的窗口
dialog --title "test" --msgbox "hello world" 10 20
```

为示例脚本16-2.sh赋予可执行权限后执行脚本，结果如图16-2所示：

```
ben@ben-laptop:~$chmod u+x 16-2.sh
ben@ben-laptop:~$./16-2.sh
```



图16-2 脚本16-2.sh执行结果

通过示例脚本16-2.sh的运行结果可以看出，在添加了--title选项以后，现实的窗口中出现了标题栏，这样可以使窗口的作用更加明显。

窗口部件可以和Bash实现数据的交互，所有部件都可以提供两种输出方式：

- 使用标准错误STDERR；
- 使用特定的退出代码状态。

dialog窗口可以将输出的内容放到标准错误STDERR中，可以使用

STDERR获取dialog输出的内容。使用STDERR可以将dialog窗口输出的内容显示到屏幕上，还可以使用重定向技术将STDERR重定向到其他文件。

dialog窗口返回的特定退出代码取决于选择的是哪一个按钮，如果选择【是】按钮或【确认】按钮，那么dialog窗口的退出代码为0，如果选择【否】按钮或【取消】按钮，那么dialog窗口的退出代码为1。关于dialog窗口的返回状态的使用如示例脚本16-3.sh和16-4.sh所示。

```
#示例脚本16-3.sh dialog命令返回值1
#!/bin/bash

echo 创建窗口，选择yes按键
dialog --yesno --title "提示框" "是否删除" 10 20
echo 返回值是$?
```

为示例脚本16-3.sh赋予可执行权限后执行脚本，结果如图16-3所示：

```
ben@ben-laptop:~$chmod u+x 16-3.sh
ben@ben-laptop:~$./16-3.sh
#示例脚本16-4.sh dialog命令返回值2
#!/bin/bash

echo 创建窗口，选择no按键
dialog --yesno --title "提示框" "是否删除" 10 20
echo 返回值是$?
```

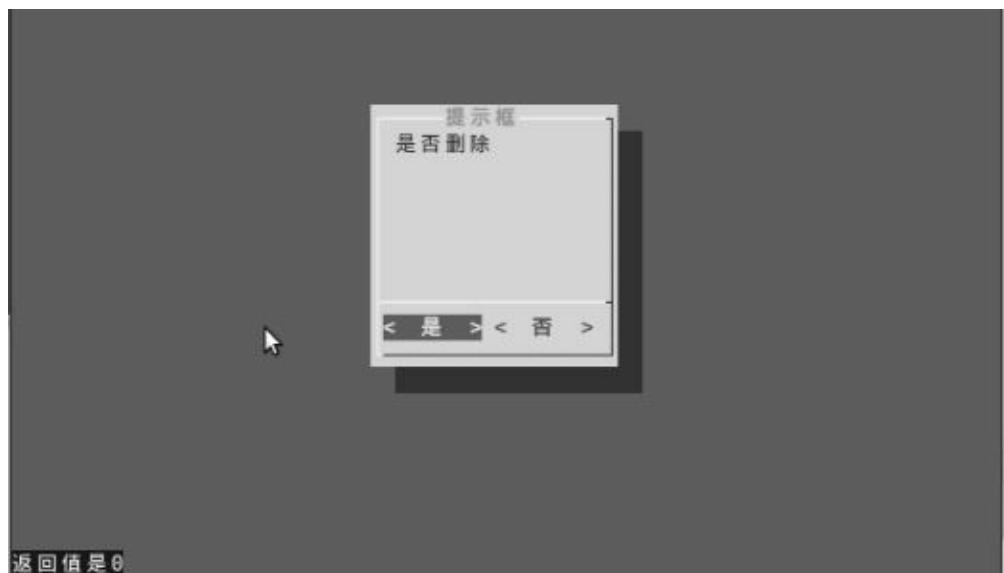


图16-3 脚本16-3.sh执行结果

为示例脚本16-4.sh赋予可执行权限后执行脚本，结果如图16-4所示：

```
ben@ben-laptop:~$chmod u+x 16-4.sh  
ben@ben-laptop:~$./16-4.sh
```



图16-4 脚本16-4.sh执行结果

通过示例脚本16-3.sh和16-4.sh的运行结果可以看出，在退出窗口时选择不同的按钮，dialog命令的退出状态就会不同，这样就可以判断在选择退出方式的时候具体选择的是哪个按钮。

注意

在bash中，一般使用标准变量\$?来获取dialog窗口的返回代码，进而获取选择的按钮。

16.2 dialog常用窗口的使用

dialog常用的窗口部件一般有消息窗口（msgbox）、yesno小窗口、文本框以及文本输入框、菜单部件等。本节将逐一介绍这些部件的使用方式。

16.2.1 消息窗口

消息窗口一般用来显示需要输出的提示性信息，在窗口上一般会有一个【确认】按钮，当单击【确认】按钮以后，该消息窗口就会退出，而该窗口的返回状态是0。使用dialog命令显示消息窗口的基本格式如下：

```
dialog --msgbox text height width
```

在上面的表达式中，text是在消息窗口中显示的内容，参数height和width表示窗口的高度和宽度，使用这两个参数以后，文本可以自动进行调整从而适应窗口的大小。在使用消息窗口时，还可以使用--title参数来表示窗口的标题。消息窗口的使用实例如示例脚本16-5.sh所示。

```
#示例脚本16-5.sh 消息窗口使用实例
#!/bin/bash

echo 创建消息窗口
dialog --title "消息窗口实例" --msgbox "显示窗口" 20 20
```

为示例脚本16-5.sh赋予可执行权限后执行脚本，结果如图16-5所示：

```
ben@ben-laptop:~$chmod u+x 16-5.sh
ben@ben-laptop:~$./16-5.sh
```



图16-5 脚本16-5.sh执行结果

通过示例脚本16-5.sh的运行结果可以看出，改变了创建窗口的尺寸之后，创建出来的窗口的大小随之变化。因此在创建窗口时，需要使得窗口的大小正合适，否则，会影响运行的整体效果。

16.2.2 yesno窗口

yesno窗口在底部有两个按钮，即【是】按钮和【否】按钮，用户可以根据实际需要选择【是】按钮或【否】按钮，如果选择【是】按钮，那么该窗口的返回状态为0，而如果选择【否】按钮，那么该窗口的返回状态为1。在选择按钮时，可以使用鼠标单击要选择的按钮，也可以使用Enter键或空格键来选择按钮。关于yesno窗口的使用如示例脚本16-6.sh所示。

```
#示例脚本16-6.sh yesno窗口使用实例
```

```
#!/bin/bash

dialog --title "是否保存" --yesno "请确认是否保存?" 10 20
```

为示例脚本16-6.sh赋予可执行权限后执行脚本，结果如图16-6所示：

```
ben@ben-laptop:~$chmod u+x 16-6.sh
ben@ben-laptop:~$./16-6.sh
```



图16-6 脚本16-6.sh执行结果

通过示例脚本16-6.sh的运行结果可以看出，yesno窗口中出现了可供选择的两个按钮：【是】和【否】。用户可以根据实际需要进行选择。

yesno窗口的默认选择按钮是【是】按钮，默认按钮就是在窗口创建以后，自动选择的按钮，默认按钮可以通过--defaultno进行修改。关

于默认按钮的选择如示例脚本16-7.sh所示。

```
#示例脚本16-7.sh yesno窗口默认按钮
#!/bin/bash

echo 修改默认按钮
dialog --title "是否保存" --defaultno --yesno "请确认是否保存?"
```

为示例脚本16-7.sh赋予可执行权限后执行脚本，结果如图16-7所示：

```
ben@ben-laptop:~$chmod u+x 16-7.sh
ben@ben-laptop:~$./16-7.sh
```

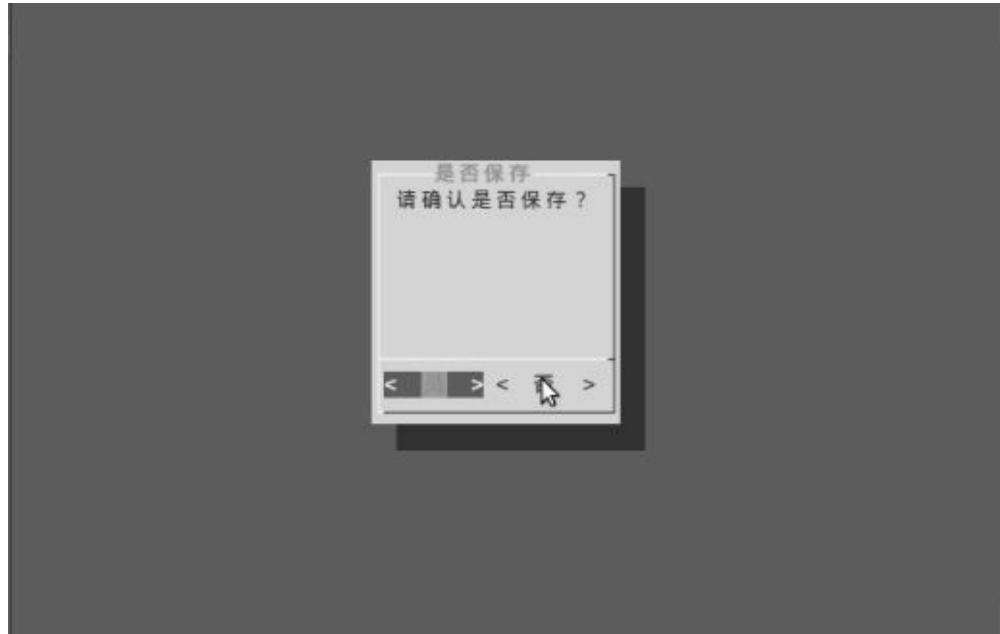


图16-7 脚本16-7.sh执行结果

通过示例脚本16-7.sh的运行结果可以看出，使用--defaultno选项

后，yesno窗口的默认选项变成了【否】按钮。通过这种方式可以改变该窗口的默认选择，这对那些大部分需要选择【否】的情形尤为适合。

16.2.3 文本框的使用

文本框一般用来输入或输出一些文本信息，常用的文本框有编辑框和文本框，在编辑框中可以输入和编辑文本，而文本框只能实现既定文本的输出，如示例脚本16-8.sh所示。

```
#示例脚本16-8.sh 使用格式控制码
#!/bin/bash

echo 显示输入框
dialog --inputbox "请输入信息" 10 20 "hello world"

echo 输入编辑框
dialog --editbox date.txt 12 30

cat date.txt
test
```

为示例脚本16-8.sh赋予可执行权限后执行脚本，结果如图16-8所示：

```
ben@ben-laptop:~$chmod u+x 16-8.sh
```

```
ben@ben-laptop:~$ ./16-8.sh
```

输入编辑框

输入编辑框

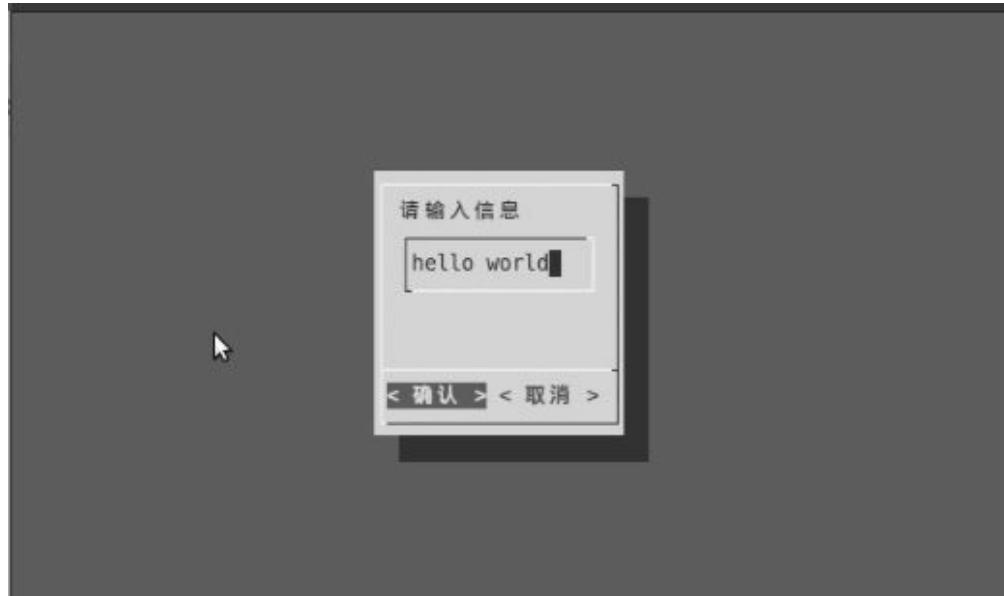


图16-8 脚本16-8.sh执行结果

接下来脚本又打开一个窗口，如图16-9所示。



图16-9 脚本16-8.sh执行结果

通过示例脚本16-8.sh的运行结果可以看出，在创建文本框时需要指定文本框的初始内容，初始内容不能被改变，而对于文本编辑框来说，可以编辑输入其他数据，当需要退出时，要使用Tab键选择【确认】或【取消】按钮。

16.2.4 菜单的使用

在使用桌面操作系统时，经常会在多个菜单中选择一个菜单的情形，这就需要提前创建菜单。创建菜单时，主要是提供一个选择标签和显示的文本。创建菜单的一般方法如下：

```
dialog --menu title height width 菜单项
```

在上面的表达式中，title表示菜单的标题，而height和width表示整个菜单窗口的高度和宽度。在添加菜单项时，基本格式如下：

```
label 文本
```

在上面的表达式中，label表示菜单的标签号，后面的文本表示菜单项中显示的内容。当用户选择相应的选项后，再单击窗口底部的【确认】按钮或【取消】按钮，然后该窗口会自动退出。只有选择了【确认】按钮以后，选中的菜单项才会生效。菜单窗口的使用如示例脚本16-9.sh所示。

```
#示例脚本16-9.sh 菜单窗口使用实例  
#! /bin/bash  
  
echo 添加菜单  
dialog --menu 选择计算方式 13 23 5 1 加法 2 减法 3 乘法 4 除法 5
```

为示例脚本16-9.sh赋予可执行权限后执行脚本，结果如图16-10所示：

```
ben@ben-laptop:~$chmod u+x 16-9.sh  
ben@ben-laptop:~$./16-9.sh
```



图16-10 脚本16-9.sh执行结果

通过示例脚本16-9.sh的运行结果可以看出，在使用了menu选项之后，可以创建供用户选择的菜单窗口，当用户选择之后就可以退出窗口

了，这样就可以记住用户的选择，便于对用户的选择进行相应的操作。

16.3 颜色的使用

在普通的操作系统中，显示的颜色是五颜六色的，而对于默认的Shell脚本来说，不管是脚本的编辑环境还是运行环境，以及运行结果的显示，都显得相对单调。因此在Shell脚本的运行过程中，可以通过设置输出格式的ANSI转义码来为Shell脚本的运行添加一点点“颜色”。

ANSI转义码以控制序列指示器（Control Sequence Indicator, CSI）开头，后面添加要在显示器上执行的操作数据，CSI的作用就是告诉终端该数据表示一个转义码，而不是普通的字符。大多数终端模拟软件都能够识别设置输出格式的ANSI转义码。

ANSI转义码可以实现多种功能，可以将光标定位在显示器的特定位置，还可以按照特定的格式来进行显示。当使用ANSI转义码时需要将对应的控制码发送到终端，需要按照一定的格式进行发送，基本格式如下：

```
^[+[+控制码+文字
```

在上面的基本格式中，第一个符号是一个ESC符号，但是一般都使用十六进制数值033表示，需要在前面加一个转移字符“\”才会启动ANSI转义码。在ESC符号的后面是一个左中括号“[”，该符号是CSI中的特定字符，和ESC按键组成整个CSI字符。控制码可以选择一个，也可以选择多个，控制码之间使用分号分隔，但是最后需要使用符号m结尾。对于一般的操作来说，使用echo命令在终端上按照特定格式来进行文字显

示和控制码重现。

在上面的结构中，m代表SGR转义码，控制显示格式的转义码也称为选择图形再现（Select Graphic Rendition，SGE）转义码，SGE转义码的基本格式如下：

```
CSI n[;k]m
```

在上面的基本格式中，n和k分别用于控制显示的内容格式。n和k可以只使用一个参数，如果都使用的话，中间需要使用分号隔开。n和k能够表示的参数内容如下：

- 效果控制代码
- 前景色控制代码
- 背景色控制代码

效果控制代码用来控制文字在终端上的显示效果，如文字的显示是普通字体还是使用斜体，显示内容的亮度是否有特殊要求，是否具有闪烁的效果，常用的效果控制代码如表16-3所示。

表16-3 显示格式使用的控制码

使用控制码	功能描述
0	重置为普通模式
1	设置为强亮度
2	设置为弱亮度

3	使用斜体
4	使用单下划线
5	使用慢闪烁
6	使用快闪烁
7	背景、前景颜色反转
8	前景色设定为背景色

通过表16-3可以看出，如果需要使用什么样的显示效果，只要使用相应的控制码即可。改变显示效果以后，还可以恢复到普通模式，如示例脚本16-10.sh所示。

```
#示例脚本16-10.sh 使用格式控制码
#!/bin/bash

echo 使用斜体显示
echo -e "\033[3mhello"
echo this is control_test

echo 使用下划线
echo -e "\033[4mhello"
echo this is control_test
```

为示例脚本16-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 16-10.sh
```

```
ben@ben-laptop:~$ ./16-10.sh
```

使用斜体显示

hello

this is control_test

使用下划线

hello

this is control test

```
ben@ben-laptop:~$
```

通过示例脚本16-10.sh的执行结果可以看出，系统使用的默认输出方式是普通模式，而在使用了不同的格式控制码以后，文字就会按照特定格式显示。而使用echo命令的-e选项是为了开启特定的显示方式，如果不使用-e选项，即使使用了格式控制码也不会按照特定的方式显示，仍然会按照默认方式显示。

在使用不管是哪种控制码改变了文字的显示方式以后，且没有恢复默认的显示格式之前，所有的输出内容都会使用上一次设定的结果进行显示，直到恢复默认显示格式之后，文字的显示方式才会使用默认方式，如示例脚本16-11.sh所示。

```
#示例脚本16-11.sh 使用格式控制码
```

```
#!/bin/bash

echo 使用下划线
echo -e "\033[4mhello"

echo 恢复默认设置
echo -e "\033[0m"
echo this is control_test
```

为示例脚本16-11.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 16-11.sh
ben@ben-laptop:~$./16-11.sh
Hello

恢复默认设置

this is control_test
ben@ben-laptop:~$
```

通过示例脚本16-11.sh的执行结果可以看出，在使用ANSI转义码以后，后续的显示内容会按照上一次成功设置的格式进行输出，直到重新设置成普通模式，输出的内容才会恢复正常。

注意

使用ANSI转义码设置了新的输出格式以后，要及时恢复默认设置，否则，后面文字显示的格式会按照上一次成功设置的格式输出。

ANSI转义码除了能够改变文字的显示格式之外，还可以控制文字显示的前景色和背景色。前景色控制代码和背景色控制代码使用同一套颜色控制代码，但是需要使用特殊的数字来标注设定的是前景色还是背景色，一般使用3来表示前景色，而背景色用4来表示。颜色控制代码如表16-4所示。

表16-4 颜色控制码

使用控制码	表示颜色
0	黑色
1	红色
2	黄色
3	绿色
4	蓝色
5	洋红色
6	青色
7	白色

在需要将特定文字以指定的颜色显示时，只需将相应的颜色控制码放到需要以该颜色显示的文字前面即可，然后当语句执行时，就会按照指定的颜色显示相应的文字，如示例脚本16-12.sh所示。

```
#示例脚本16-12.sh 格式控制码的简单使用
#!/bin/bash

echo 背景色变成红色
echo -e "\033[41mhello\033[0m"

echo 前景色变成黄色
echo -e "\033[32mhello\033[0m"
```

为示例脚本16-12.sh赋予可执行权限后执行脚本，结果如图16-11所示：

```
ben@ben-laptop:~$chmod u+x 16-12.sh
ben@ben-laptop:~$./16-12.sh
```

```
ben@ben-laptop:~/桌面$ ./16-12.sh
背景色变成红色
hello
前景色变成黄色
hello
```

图16-11 脚本16-12.sh执行结果

通过示例脚本16-12.sh的执行结果可以看出，在使用了ANSI控制码以后，会按照特定格式输出文字。为了保证后续输出能够正常显示，要在输出的最后恢复输出格式为默认格式。

注意

对于一般情况来说，能引起操作者注意的内容一般是在菜单栏中选择或其他需要用户输入的地方，或者是输出内容中比较重要的地方。

16.4 创建菜单

菜单栏在Windows系统以及Linux系统中经常出现，一般会将所有可能出现的选项放到一起，作为一个菜单，而使用者需要做的就是按照自己的实际需求，从所有的菜单项中选择适合自己的项目。而在Shell脚本中也可以实现类似的功能。本节就通过模拟实现计算器的基本功能来展示菜单的创建和使用。

16.4.1 在Shell脚本中创建菜单

在Shell脚本中创建菜单，可以使用dialog命令中的--menu选项来完成，还可以使用case命令。本小节主要介绍如何使用case命令创建菜单。

一般使用echo命令来显示需要显示的菜单项，在显示菜单项时，一般是将菜单和内容分开，这样方便用户选择，因此需要使用-e选项使得在语句中出现的特殊字符（如换行符、制表符等）被作为特殊字符处理。使用echo命令创建菜单项的示例脚本如16-13.sh所示。

```
#示例脚本16-13.sh 使用echo命令创建菜单

#!/bin/bash

echo 请按照菜单项选择进行操作的方式
echo -n "1 加法"
echo "2 减法"
echo -n "3 乘法"
echo "4 除法"
echo "5 退出"
```

为示例脚本16-13.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 16-13.sh
ben@ben-laptop:~$./16-13.sh
请按照菜单项选择进行操作的方式
1 加法    2 减法
3 乘法    4 除法
5 退出
```

当菜单栏显示到屏幕上以后，用户要根据需要来进行选择。选择时一般是使用数字或单个字符，因此可以使用read命令来获取用户的输入。为了保证用户输入的是单个字符，还需要使用-n选项来进行限定。当使用-n选项时，不需要按Enter键，当用户输入一个字符以后，自动将输入的字符赋值给变量。read命令的使用方式如示例脚本16-14.sh所示。

```
#示例脚本16-14.sh 输入操作方式
```

```
#!/bin/bash

echo -n 请输入选项方式:
read -n choose
echo 用户选择的是: $choose
```

为示例脚本16-14.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 16-14.sh
ben@ben-laptop:~$./16-14.sh
请输入选项方式: 1
用户选择的是: 1
```

示例脚本16-13.sh和16-14.sh的作用是对用户进行友好性提示，提示用户当前应该进行何种操作，并且获取用户输入的操作方式以及两个操作数，从而为后续的操作做必要的数据准备。

16.4.2 创建子菜单函数

在高级编程语言中，一般使用函数来实现各个子模块。比如整个菜单栏的显示、子菜单功能的实现等，都可以使用函数来实现。函数可以将某些比较独立的功能封装到一起，第9章已经介绍了如何使用函数，现在就利用第9章中的知识来解决这个实际问题，即使用函数实现加减乘除4种运算。

在计算之前，需要首先输入两个数值作为操作数据，输入整数在示

例脚本中即使用read命令为两个变量赋值，如示例脚本16-15.sh所示。

```
#示例脚本16-15.sh 输入两个操作数
#!/bin/bash

echo -n 请输入第一个操作数:
read num1

echo -n 请输入第二个操作数:
read num2

echo 输入的操作数为$num1和$num2
```

为示例脚本16-15.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 16-15.sh
ben@ben-laptop:~$./16-15.sh
请输入第一个操作数: 10
请输入第二个操作数: 14
输入的操作数为10和14
```

对于每一个子菜单来说，都可以使用一个子函数来实现。这些子函数可以通过case命令连接在一起，从而构成整个程序的主体部分。case命令的作用就是根据从菜单选择的字符调用该字符对应的菜单，类似于高级语言中的switch结构。关于case命令的使用如示例脚本16-16.sh所示。

```
#示例脚本16-16.sh 数据操作部分
#!/bin/bash

case $choose in
1)echo $num1 + $num2 = $((num1+num2));;
2)echo $num1 - $num2 = $((num1-num2));;
3)echo $num1 * $num2 = $((num1*num2));;
4)echo $num1 / $num2 = $((num1/num2));;
5)return 0;;
*)echo 输入错误
esac;;
```

通过16-16.sh可以看出，根据输入的操作方式选择case结构中不同的分支，可实现不同的操作方式。

对于示例脚本16-16.sh来说，除了可以使用case结构之外，还可以使用if-then-else结构来实现运用不同操作符进行计算的算法，如示例脚本16-16-1.sh所示。

```
#示例脚本16-16-1.sh 使用if结构代替case结构
#!/bin/bash

if [ $choose -eq 1 ]
then
    echo $num1 + $num2 = $((num1+num2))
elif [ $choose -eq 2 ]
then
```

```
echo $num1 - $num2 = $($((num1-num2)))
elif [ $choose -eq 3 ]
then
    echo $num1 * $num2 = $($((num1*num2)))
elif [ $choose -eq 4 ]
then
    echo $num1 / $num2 = $($((num1/num2)))
else
then
    echo 输入错误
fi
```

16.4.3 脚本的整合

通过前面两个小节的讲述，计算器程序的Shell脚本已经初具规模，剩下的工作就是将上面相关的脚本按照流程进行整合，从而满足整体的需要。该计算器的设计思路如图16-12所示。

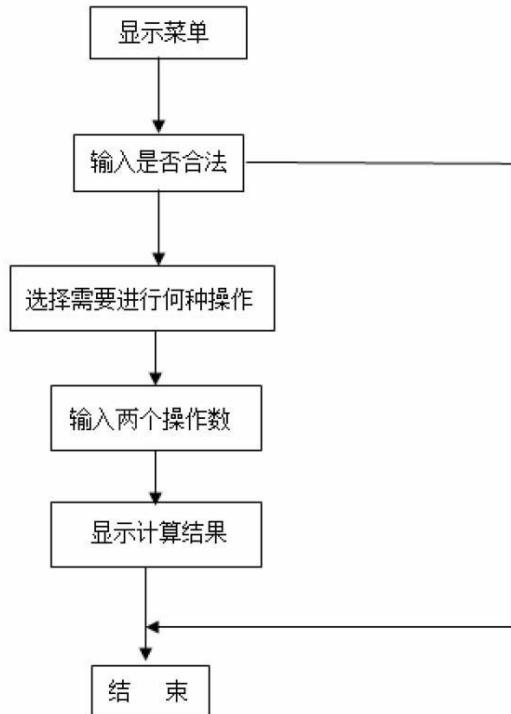


图16-12 模拟计算器操作流程

通过图16-1可以看出，在脚本执行以后，首先要将菜单显示出来，从而可使用户选择要进行的操作。当获取用户选择的操作符后，判断用户的选择是否正确，只有正确的选择才可以进行后面的操作。如果用户选择错误，则结束脚本的运行。选择了正确的操作以后，根据提示输入用于操作的两个操作数，然后，根据用户选择的操作以及输入的操作数，进行计算，并将结果显示出来。具体的内容如示例脚本16-17.sh所示。

```
#示例脚本16-17.sh 脚本的整合
```

```
#!/bin/bash
```

```
16-15.sh
```

```
16-13.sh
```

16-14.sh

16-16.sh

为示例脚本16-17.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 16-17.sh
```

```
ben@ben-laptop:~$./16-17.sh
```

请输入第一个操作数： 10

请输入第二个操作数： 14

输入的操作数为10和14

请按照菜单项选择进行操作的方式

1 加法 2 减法

3 乘法 4 除法

5 退出

请输入选项方式： 1

用户选择的是： 1

10+14=24

通过示例脚本16-17.sh的执行结果可以看出，使用echo命令能够按照菜单的方式进行输出，而使用case结构可以根据用户的输入方式来进行相应的处理。

注意

脚本16-17.sh还可以使用循环结构来使整个操作过程循环地进行，从而使用户能够进行多次操作。

16.5 小结

本章主要介绍如何优化Shell脚本，主要包括添加窗口、给Shell脚本的运行增加颜色、创建菜单等3部分内容。这3部分内容使得Shell脚本的运行更加“绚丽多彩”。

dialog软件包是Linux系统中常用的软件包，在Linux系统中可以使用命令在Shell脚本中创建窗口，可以通过选项以及窗口类型来创建各种类型的窗口，并且可以使用各种参数来调整创建的窗口的大小以及展示的内容，从而使得Shell脚本更加生动。

使用ANSI转义码可以发送转义码到终端，从而使得脚本运行时不再是单调的颜色，而可以采用“五颜六色”的展示方式表示在终端中。

在Shell脚本中，可以使用case命令创建一个菜单，然后根据用户的选择来执行对应菜单的功能项。每一个子菜单项都可以使用函数来实现。这样可以创建一个具有交互性的Shell脚本。还可以使用select命令代替case命令直接根据用户的输入来处理对应的功能模块。

第17章 Shell实战之系统管理

对于任何一个系统来说，对系统进行管理都是必要的。对于Shell脚本来说，每一个脚本均由bash命令构成，其执行效率是非常高的，而且包含很多系统管理的命令，因此可以使用Shell脚本非常简便地实现系统的管理。本章就重点介绍如何使用Shell脚本实现Linux系统的管理。

本章的主要内容如下：

- 系统监测脚本的编写
- 计划任务的实现
- 网络管理
- 日志管理

17.1 系统监测

对于Linux系统来说，系统监控是必需的。在系统运行过程中，会发生各种各样的意外情况，如内存耗尽、存储空间耗尽等，这些情况严重影响系统的正常运行。系统监控用来监控系统的运行状况，从而及时发现系统运行时的问题，本节就介绍如何对Linux系统进行监测。

17.1.1 系统监控基础

系统监控的基础主要来源有以下两种：

- 系统文件
- 系统管理命令

系统文件记录了系统运行时当前的所有信息。这些文件都在目录/proc中，分别记录了系统运行时的CPU、内存、硬盘空间等系统信息，于是可以通过读取这些文件中相应的字段来获取需要的系统信息。而系统管理相关的命令在前面的章节中已经介绍过，在此不再赘述。本小节主要介绍配置文件的相关内容。

注意

用于Linux系统管理的命令大部分也是通过获取相应文件中的内容以后，再通过特定的格式来实现系统信息的输出。

对于Linux系统来说，CPU资源是非常重要的资源，其相关信息存储在文件/proc/cpuinfo中，该文件的内容如下：

```
ben@ben-laptop:~$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model
      : 37
model name : Intel(R) Core(TM) i5 CPU        M 450 @ 2.40GHz
stepping : 5
cpu MHz
```

```
: 1199.000
cache size : 3072 KB
physical id : 0
siblings : 4
core id
: 0
cpu cores : 2
apicid
: 0
initial apicid : 0
fdiv_bug : no
hlt_bug
: no
f00f_bug : no
coma_bug : no
fpu
: yes
fpu_exception : yes
cpuid level : 11
wp
: yes
flags
: fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov
bogomips : 4788.63
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
```

```
power management:

processor : 1
vendor_id : GenuineIntel
cpu family : 6
model
: 37
model name : Intel(R) Core(TM) i5 CPU      M 450 @ 2.40GHz
stepping : 5
cpu MHz
: 1199.000
cache size : 3072 KB
physical id : 0
siblings : 4
core id
: 0
cpu cores : 2
apicid
: 1
initial apicid : 1
fdiv_bug : no
hlt_bug
: no
f00f_bug : no
coma_bug : no
fpu
: yes
```

```
fpu_exception : yes
cpuid level : 11
wp
: yes
flags
: fpu vme de pse tsc msr pae mce cx8 apic mttr pge mca cmov
bogomips : 4787.88
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management:
```

在该文件中存储了CPU的一些信息，而这里显示了CPU的基本信息，如CPU的内核版本型号、主频、缓存等信息，计算机中的CPU是几个内核，在该文件中就有几部分内容与之对应。

还可以使用top命令来获取基本的CPU信息，除了显示系统中进程占用的CPU资源信息之外，还能显示空闲CPU的比例，如示例脚本17-1.sh所示。

```
#示例脚本17-1.sh  显示CPU资源信息
#!/bin/bash

echo 显示CPU空闲比例
idle=`top -b -n 1 | grep Cpu | gawk '{print $5}' | cut -f 1
echo CPU当前的空闲比例为$idle
```

为示例脚本17-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-1.sh  
ben@ben-laptop:~$./17-1.sh  
显示CPU空闲比例  
CPU当前的空闲比例为92
```

在示例脚本17-1.sh中使用了grep命令和gawk命令以及cut命令，从top命令的显示结果中获取了CPU当前的空闲比例，当CPU空闲比例小于一定值时，就需要采取一定处理方式，防止因CPU使用率过高而引起系统运行故障。

对于内存信息和硬盘空间信息来说，可以从配置文件中获取，也可以通过命令获取。内存信息对应的配置文件为/etc/meminfo，而存储硬盘空间相关信息的配置文件是/etc/，这两个文件的内容如下：

```
root@ben-laptop:~# cat /proc/meminfo  
MemTotal:           2569116 kB  
MemFree:            591300 kB  
Buffers:             253500 kB  
Cached:              912744 kB  
SwapCached:          0 kB  
Active:              956652 kB  
Inactive:            722432 kB  
Active(anon):       486408 kB  
Inactive(anon):     41276 kB  
Active(file):        470244 kB
```

```
Inactive(file):      681156 kB
Unevictable:          16 kB
Mlocked:              16 kB
HighTotal:            1702152 kB
HighFree:             142848 kB
LowTotal:             866964 kB
LowFree:              448452 kB
SwapTotal:            1999864 kB
SwapFree:             1999864 kB
Dirty:                 48 kB
Writeback:              0 kB
AnonPages:            512872 kB
Mapped:               103680 kB
Shmem:                14844 kB
Slab:                 113672 kB
SReclaimable:         99144 kB
SUnreclaim:           14528 kB
KernelStack:          2952 kB
PageTables:           8772 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:           0 kB
CommitLimit:           3284420 kB
Committed_AS:          1414976 kB
VmallocTotal:           122880 kB
VmallocUsed:            38580 kB
VmallocChunk:           77336 kB
```

```
HardwareCorrupted:      0 kB
HugePages_Total:        0
HugePages_Free:         0
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:           4096 kB
DirectMap4k:             12280 kB
DirectMap4M:             897024 kB
```

上面所说的配置文件中记录了系统的运行信息，而进行监控的主要数据来源就是这些配置文件。用户可以根据自己的需要读取不同的配置文件来获取相应的数据。

17.1.2 Ubuntu自带的系统监控工具

在Ubuntu系统中，提供了很多可以用于系统监控的操作方式，如在菜单【系统管理】中，提供了【系统监视器】和【系统日志查看器】两个菜单。通过这两个菜单可以实现日常的部分监控操作。

依次打开【系统】|【系统管理】|【系统监视器】，就可以查看系统日志。在【系统监视器】菜单中提供了很多可用于系统监视的选项。可以查看系统运行情况、进程运行情况、资源使用历史以及文件系统的使用等。打开该菜单以后，其内容如图17-1所示。

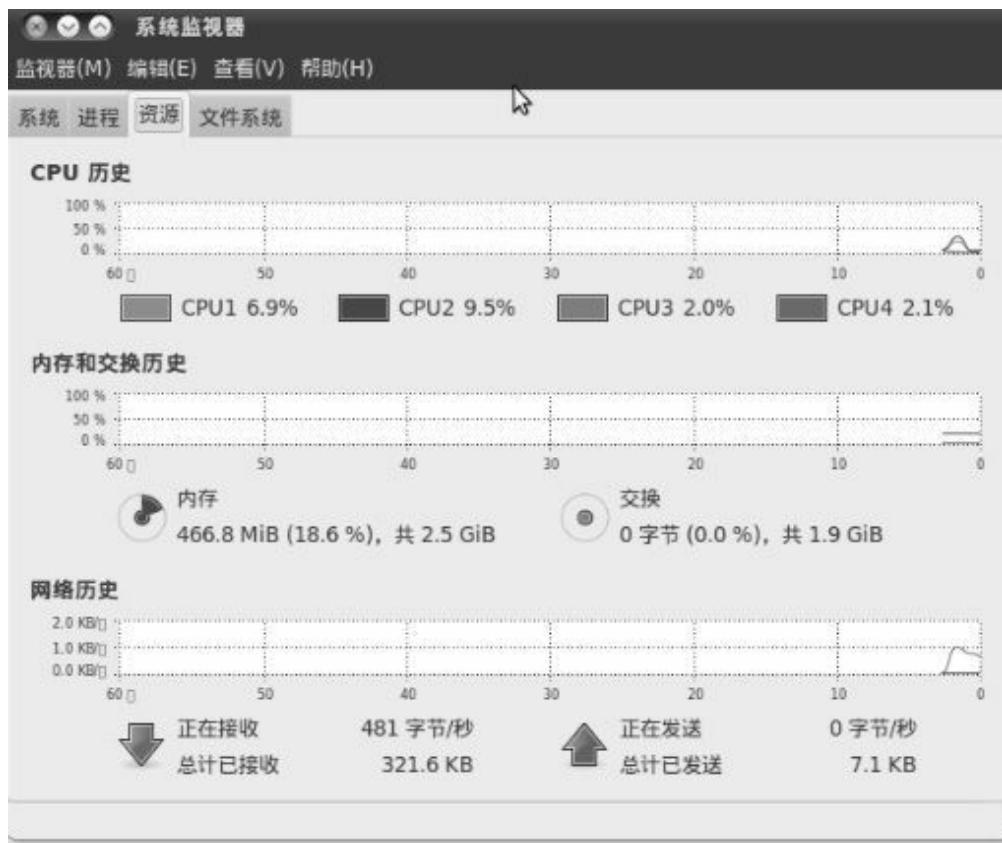


图17-1 【系统监视器】显示界面

通过【系统监视器】，可以非常直观地了解Linux系统运行时的使用情况。

使用【日志查看器】可以查看Linux系统日志信息，依次打开【系统】|【系统管理】|【系统日志查看器】，就可以查看系统日志。打开【系统日志查看器】以后，其界面显示如图17-2所示。

```

文件(F) 编辑(E) 查看(V) 过滤器(F) 帮助(H)
X.org.0.log Jul 28 21:23:74 ben-laptop kernel: Kernel logging (proc) stopped.
X.org.1.log Jul 29 23:12:44 ben-laptop kernel: imklog 4.2.0, log source = /proc/kmsg started.
X.org.2.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] Initializing cgroup subsys cpuset
+ auth.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] Initializing cgroup subsys cpu
+ auth.log.1 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] KERNEL supported cpus:
boot Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] Intel GenuineIntel
boot.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] AMD AuthenticAMD
bootstrap.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] NSC Geode by NSC
+ daemon.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] Cyrix CyrixInstead
+ daemon.log.1 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] Centaur CentaurHauls
+ debug Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] Transmeta GenuineTMx86
+ debug.1 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] Transmeta TransmetaCPU
+ dmesg Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] UMC UMC UMC
+ dmesg.0 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-provided physical RAM map:
+ dmesg.1 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 0000000000000000 - 000000000000d000 (usable)
+ dpkg.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 0000000000009d000 - 000000000000a0000 (reserved)
+ dpkg.log.1 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 00000000000e0000 - 00000000000100000 (reserved)
+ fontconfig.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 00000000000f63f000 (usable)
+ jockey.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 00000000000f63f000 (reserved)
+ jockey.log.1 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 00000000000f63f000 (reserved)
+ kern.log Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 00000000000fec00000 - 00000000000fec10000 (reserved)
+ kern.log.1 Jul 29 23:12:44 ben-laptop kernel: [ 0.000000] BIOS-e820: 00000000000fed10000 - 00000000000fed14000 (reserved)

```

图17-2 【系统日志查看器】显示界面

由图17-2可以看出，在【系统日志查看器】中可以看到所有的系统日志信息，如内核信息以及启动信息等。可以通过该界面查看系统日志，从而作为对系统管理的参考信息。

注意

除系统自带的图形化监控工具之外，还有很多第三方的监控工具，用户可以选择安装合适的软件进行系统监控。

17.1.3 监控脚本的编写

显示内存信息和硬盘空间信息的命令分别是free命令和df命令。而监控脚本中信息主要是来自于这两个命令的运行结果，使用gawk命令和grep命令将对需要的内容进行匹配，从而得到需要的信息，如示例脚本17-2.sh所示。

```
#示例脚本17-2.sh 监控脚本示例

#!/bin/bash


echo 获取剩余交换空间
swap_free=`free -m | grep Swap | gawk '{print $4}'` 
if (( swap_free < 15 ))
then
    echo 交换空间剩余不足
    fi
echo 交换空间还剩$swap_free
echo


echo 获取剩余普通内存空间
mem_free=`free -m | grep Mem | gawk '{print $4}'` 
if (( mem_free < 15 ))
then
    echo 内存空间剩余不足
    fi
echo 内存空间还剩$mem_free
echo


echo 获取剩余硬盘空间
diska_free=`df -h | grep /dev/sda6 | awk '{print $5}' | cut -c 1-3` 
if ((diska_free < 15 ))
then
    echo "磁盘/dev/sda6空间剩余不足"
```

```
fi  
echo "磁盘/dev/sda$i 的剩余空间为$diska_free"
```

为示例脚本17-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-2.sh  
ben@ben-laptop:~$./17-2.sh  
获取剩余交换空间  
交换空间还剩1952  
  
获取剩余普通内存空间  
  
内存空间还剩356  
  
获取剩余硬盘空间  
磁盘/dev/sda 的剩余空间为34
```

通过示例脚本17-2.sh的执行结果可以看出，使用脚本可以非常简单地监控Linux系统的资源使用情况，从而获取对系统进行处理的依据。

17.2 计划任务的实现

在实际操作过程中，经常会遇到每隔一段时间就需要执行某个脚本，或者在每个月或每天的固定时间执行某些脚本的情况。这些都是计划任务的范畴。在Linux系统中，计划任务可以依靠at命令和cron命令来实现。本节重点介绍如何使用这两个命令来实现计划任务。

17.2.1 at命令的使用

在Linux系统中，计划任务有的执行一次，而有的则需要周期性地执行多次。一次性的定时计划就是该计划执行一次，当计划执行完以后，下次执行的时间不确定，并且每次都需要手动启动。对于周期性计划来说，当计划执行完以后，在固定的时间内还会执行下一次，直至计划被删除，否则将按照执行周期循环执行。at命令一般用来处理一次性计划，其使用方式如下：

```
at [选项/参数] [时间]
```

at命令在使用时要添加适当的参数或选项，从而使得该计划的执行细节能够被确定。常用的参数如表17-1所示。

表17-1 at命令的常用参数

选项	作用
-m	当指定的任务被完成之后，将给用户发送邮件，即使没有标准输出
-I	atq的别名
-d	atrm的别名
-v	显示任务将被执行的时间
-c	打印任务的内容到标准输出
-V (大写字符)	显示版本信息

-q <队列>	使用指定的列队
-f <文件>	从指定文件读入任务而不是从标准输入读入
-t <时间参数>	以时间参数的形式提交要运行的任务

at命令能够实现一次性计划的设定，当提交了一个计划任务以后，在系统中就会启动一个atd进程，这个atd进程是守护进程，它不停地在文件/var/spool/at中扫描是否存在需要处理的任务，如果任务的处理时间符合处理要求，那么该计划任务就会被执行。如果对at命令不是很熟悉，可以使用-help选项查看at命令的帮助，如下所示：

```
ben@ben-laptop:~$ at -help
Usage: at [-V] [-q x] [-f file] [-mldbv] timespec ...
        at [-V] [-q x] [-f file] [-mldbv] -t time
        at -c job ..
        atq [-V] [-q x]
        atrm [-V] job ..
        batch
```

如果atd进程没有启动，那么at命令就不会被执行。可以使用grep命令检索atd命令是否已经执行。Grep命令的使用方式如下：

```
ps -ef | grep atd | grep -v grep
```

执行结果如下：

```
ps -ef | grep atd | grep -v grep
```

```
daemon      1217      1  0 22:39 ?          00:00:00 atd
```

如果没有任何显示，则说明atd没有执行。如果没有执行，就先启动atd命令。启动atd命令如下：

- /etc/init.d/atd start
- /etc/init.d/atd restart

在确保atd命令执行的情况下，at命令会从标准输入STDIN获取输入信息并提交到计划队列中，然后写入文件/var/spool/at，启动守护进程atd后，就等待计划在规定的时间内执行。除此之外，还可以通过文件传递给计划任务需要执行的命令，如示例脚本17-3.sh所示。

```
#示例脚本17-3.sh 使用at命令创建计划任务
```

```
#!/bin/bash

echo 使用at命令创建计划任务
at -f 17-3.sh now + 2 minutes
echo

echo 显示设定的定时任务
at -l
```

为示例脚本17-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-3.sh
ben@ben-laptop:~$./17-3.sh
```

```
使用at命令创建计划任务
```

```
warning: commands will be executed using /bin/sh
job 1 at Sun Jul 27 20:04:00 2014
```

```
显示设定的定时任务
```

```
1 Sun Jul 27 20:04:00 2014 a ben
```

通过示例脚本17-3.sh的执行结果可以看出，使用了at命令之后，就创建了一个定时任务，而使用at命令的特定选项可以显示已经存在的任务。at命令在运行之后会生成一条警告信息，该信息中包含使用哪种Shell来执行这个定时任务，并且显示出任务执行的具体时间和任务编号。

at命令可以使用多种时间形式来确定计划任务的执行时间，常用的表示时间的方式如下：

- 标准时间格式，包括小时分钟和标准的日期格式；
- 12小时制和24小时制；
- 具体的时间；
- 文本类型的日期格式；
- 指定时间增量。

在上面的时间格式中，标准的时间格式就是标准的小时分钟格式和标准的日期格式，如凌晨1点钟可以使用1:00表示，而日期可以使用下列基本格式表示：

- YYYY/MM/DD；
- MM/DD/ YY；

- DD. MM. YY。

在上面的格式中，YY（或YYYY）一般用来表示年份，MM用来表示月份，而DD用来表示是哪一天，如2014年2月14日，可以使用下面的3种形式来表示：

```
2014/02/14
```

```
02/14/2014
```

```
14.02.2014
```

在为计划任务指定时间时，可以使用标准的24小时制，也可以使用12小时制进行设定。在使用12小时制时，需要使用指示符PM或AM来指示上午还是下午，如表示下午4点20分即16点20分，使用24小时制表示为16:20，使用12小时制表示为4:20PM。关于这两种时间的表示方式如示例脚本17-4.sh所示。

```
#示例脚本17-4.sh 使用at命令创建计划任务
```

```
#!/bin/bash
```

```
echo 使用12小时制设定计划任务
```

```
at -f 17-4.sh 4:20PM tomorrow
```

```
echo
```

```
echo 使用24小时制设定计划任务
```

```
at -f 17-4.sh 16:20 tomorrow
```

```
echo
```

```
echo 显示设定的定时任务
```

```
at -l
```

为示例脚本17-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-4.sh
```

```
ben@ben-laptop:~$./17-4.sh
```

使用12小时制设定计划任务

```
warning: commands will be executed using /bin/sh
```

```
job 2 at Mon Jul 28 4:20PM 2014
```

使用24小时制设定计划任务

```
warning: commands will be executed using /bin/sh
```

```
job 3 at Mon Jul 28 16:20:00 2014
```

显示设定的定时任务

```
1 Sun Jul 27 20:04:00 2014 a ben
```

```
2 Mon Jul 28 4:20PM 2014 a ben
```

```
3 Mon Jul 28 16:20:00 2014 a ben
```

除了使用标准时间以外，at命令还支持一些具体的时间以及文本类型时间，如noon、now、midnight、teatime等，这些单词都表示具体的时间，对于teatime来说，一般认为是下午4点即16点。除此之外，还可以使用文本（如OCT 10或DEC 24等）表示具体的时间，如示例脚本17-5.sh所示。

```
#实例脚本17-5.sh    其他时间类型  
#! /bin/bash  
  
echo 使用文本型时间  
at -f 17-4.sh midnight  
echo  
  
at -f 17-4.sh teatime  
echo
```

为示例脚本17-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-5.sh  
ben@ben-laptop:~$./17-5.sh  
使用文本型时间  
warning: commands will be executed using /bin/sh  
job 4 at Mon Jul 28 00:00:00 2014  
  
warning: commands will be executed using /bin/sh  
job 5 at Mon Jul 28 16:00:00 2014
```

at命令还可以使用时间增量的方式来确定时间，如从现在开始往后数2天或2小时，从2014年2月14日往后数10天等。可以使用符号“+”或文本tomorrow等来表示增加到什么时候。关于时间增量的使用方式如示例脚本17-6.sh所示。

```
#示例脚本17-6.sh    使用时间增量
#!/bin/bash

echo 使用时间增量完成任务的设定
at -f 17-4.sh midnight +2 hours
echo

at -f 17-4.sh teatime + 2hours
```

为示例脚本17-6.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-6.sh
ben@ben-laptop:~$./17-6.sh
使用时间增量完成任务的设定
warning: commands will be executed using /bin/sh
job 6 at Mon Jul 28 02:00:00 2014

warning: commands will be executed using /bin/sh
job 7 at Mon Jul 28 18:00:00 2014
```

通过上面的示例脚本可以看出，使用at命令可以设定各种定时任务。在设定定时任务时，会按照设定的执行时间执行特定的命令。而设定的执行时间可以使用各种方式表示。

at命令产生的计划任务都会放到任务队列中，守护进程atd会到相应的任务队列中去查找有没有合适的计划任务被执行。在Linux系统中，可以存在26个任务队列，分别使用小写字符a、b、c.....z表示。其中a

级任务队列中的任务优先级最高，而at命令默认将任务放到a级队列中，从而使得计划任务能够在第一时间被执行。而如果任务不是很紧急，也可以放到其他队列中。使用-q选项可指定将任务放到哪个队列中。关于任务队列的实例如示例脚本17-7.sh所示。

```
#示例脚本17-7.sh 指定任务队列
#!/bin/bash

echo 使用-q选项指定任务队列
at -f 17-4.sh midnight +2 hours -q b
```

为示例脚本17-7.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-7.sh
ben@ben-laptop:~$./17-7.sh
使用-q选项指定任务队列
warning: commands will be executed using /bin/sh
job 8 at Mon Jul 28 02:00:00 2014
```

计划任务在任务队列中等待运行时，如果临时改变计划，还可以将计划删除。删除计划任务使用命令atrm，还可以使用atq命令来查看当前的任务。关于这两个命令的使用方式如示例脚本17-8.sh所示。

```
#示例脚本17-8.sh 任务的删除
#!/bin/bash
```

```
echo显示设定的任务
```

```
at -l
```

```
echo
```

```
echo 删除任务号为3的任务
```

```
atrm 3
```

```
echo
```

```
echo重新显示设定的任务
```

```
at -l
```

为示例脚本17-8.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-8.sh
```

```
ben@ben-laptop:~$./17-8.sh
```

```
显示设定的任务
```

```
1 Sun Jul 27 20:04:00 2014 a ben
```

```
6 Mon Jul 28 02:00:00 2014 a ben
```

```
3 Mon Jul 28 16:20:00 2014 a ben
```

```
8 Mon Jul 28 02:00:00 2014 b ben
```

```
2 Mon Jul 28 4:20:PM 2014 a ben
```

```
删除任务号为3的任务
```

```
重新显示设定的任务
```

```
1 Sun Jul 27 20:04:00 2014 a ben
```

```
6     Mon Jul 28 02:00:00 2014 a ben
8     Mon Jul 28 02:00:00 2014 b ben
2 Mon Jul 28 4:20PM 2014 a ben
```

通过示例脚本17-8.sh的执行结果可以看出，使用at命令添加定时任务时，可以指定个各种时间、指定操作的队列。当任务建立以后，还可以使用atrm命令删除特定的任务，任务删除之后就不会再执行了。

17.2.2 atq命令的使用

在上面的小节中展示任务队列时，使用了at命令的-l选项，除此之外，还可以使用atq命令来显示这些队列。atq命令的主要作用就是列出用户排在队列中的作业，其使用方式如下：

```
atq [选项]
```

atq命令在执行时，需要配合使用选项，从而确定需要显示哪些内容，常用的选项如下：

- -q：显示某个队列中的任务。
- -V：显示atq程序的版本信息。

在一般情况下，直接使用atq命令即能实现命令at -l的效果，atq命令的使用方式如示例脚本17-9.sh所示。

```
#示例脚本17-9.sh    atq命令的使用
```

```
#!/bin/bash
```

echo 显示设定的任务

```
atq
```

```
echo
```

echo 显示a队列中的任务

```
atq -q a
```

```
echo
```

echo 显示b队列中的任务

```
atq -q b
```

为示例脚本17-9.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-9.sh
```

```
ben@ben-laptop:~$./17-9.sh
```

显示设定的任务

```
1 Sun Jul 27 20:04:00 2014 a ben
```

```
6 Mon Jul 28 02:00:00 2014 a ben
```

```
3 Mon Jul 28 16:20:00 2014 a ben
```

```
8 Mon Jul 28 02:00:00 2014 b ben
```

```
2 Mon Jul 28 4:20PM 2014 a ben
```

显示a队列中的任务

```
1 Sun Jul 27 20:04:00 2014 a ben
```

```
6      Mon Jul 28 02:00:00 2014 a ben  
3 Sun Mon Jul 28 16:20:00 2014 a ben  
2 Mon Jul 28 4:20PM 2014 a ben
```

显示b队列中的任务

```
8      Mon Jul 28 02:00:00 2014 b ben
```

通过示例脚本17-9.sh的执行结果可以看出，使用atq命令可以展示出正在等待执行的任务，还能指定显示具体队列中的任务。

17.2.3 cron的使用

at命令创建的计划任务是一次性的，当执行完一次以后，就不会再执行，除非再次生成这个任务。而在实际使用中，会经常出现需周期性地执行某个任务的情况，如果使用at命令，就需要不断执行at命令来保证计划任务能够被执行。在Linux系统中，可以使用cron命令来实现周期性任务的执行。

cron程序一般是从后台启动，然后从cron表格（cron table）中查找能够执行的计划任务。cron表格使用特殊格式指定计划任务执行时间，后台程序根据任务的执行时间来确定当前需要执行哪些任务。cron表格的基本格式如下：

min	hour	dayofmonth	month	dayofweek	commands
-----	------	------------	-------	-----------	----------

这个语句用来表示在哪一月的哪一天或是哪一周的哪一天的几点几

分执行什么命令，其中，dayofmonth指定每个月的日期值，其范围为1~31；而dayofweek可以使用两种方式表示，一种是常用的英文单词的缩写形式，即用mon、tue、wed、thu、fri、sat、sun分别表示周一至周日。除此之外，还可以使用数字0到6分别表示周日、一……六，其中0表示周日，而1表示周一。所有字段的内容可以指定为具体值，也可以使用范围或通配符等。当使用通配符时，如在hour的位置使用了通配符，表示每个小时都要执行。

cron计划任务通过cron表格来实现任务周期性的定时执行，cron表格由crontab命令来处理，一般存放在目录文件/var/spool/cron/crontabs中，文件名和用户名是相同的，在文件中展示了所有定时任务。可以直接向该文件中写入需要定时执行的内容，而不使用crontab命令也可以实现同样的效果。

操作cron表格时一般使用crontab命令，crontab命令的常用选项如下：

- -e选项：向cron表格中添加条目。
- -l选项：显示出所有的cron表格。

crontab命令的使用方式如示例脚本17-10.sh所示。

```
#示例脚本17-10.sh    任务的删除
#!/bin/bash

echo显示设定的任务
crontab -l
```

为示例脚本17-10.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-10.sh  
ben@ben-laptop:~$./17-10.sh  
* * * * * ls -l > log.txt  
* * * * * ls -l > log1.txt
```

示例脚本17-10.sh的执行结果中展示的是早就添加到cron表格中的内容，如果不对配置文件进行编辑，还可以使用crontab命令进行添加。使用crontab命令添加cron定时任务时，需要使用-e选项，然后开始编辑cron表格，如图17-3所示。

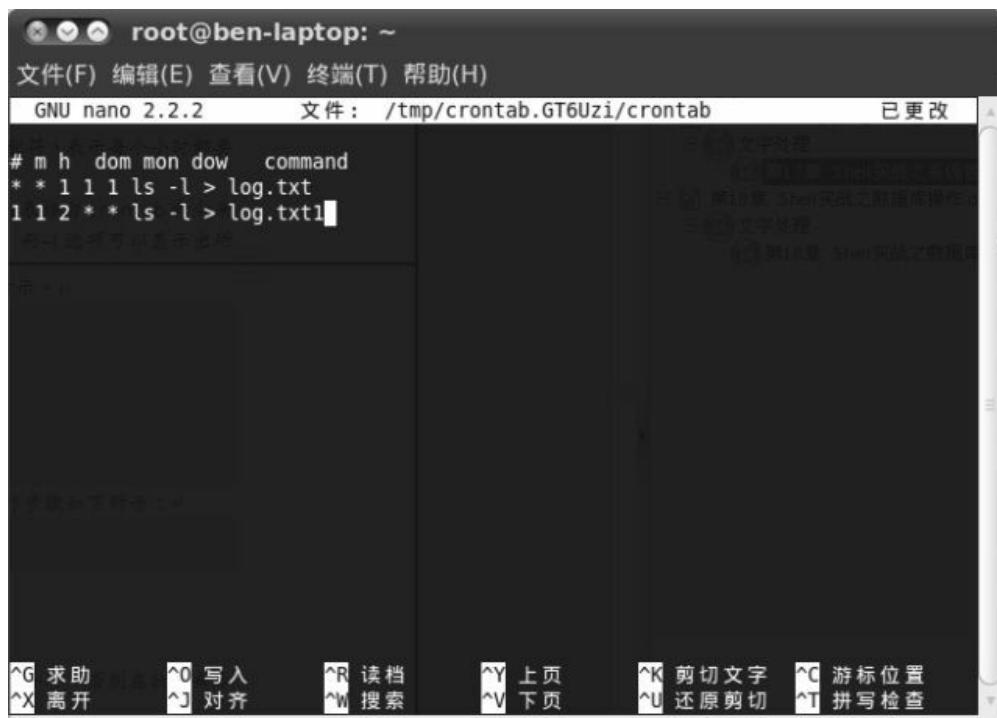


图17-3 编辑cron表格

在编辑完cron表格以后，按照下面的提示行操作就可以退出编辑窗口，从而将编辑的内容写入配置文件。

注意

根据默认约定，`cron`表格是不存在的，在使用之前，必须先创建它。否则在执行命令时，会提示没有`crontab`。

17.3 网络管理

对于Linux系统来说，网络模块的管理也是重要的部分，本节将重点介绍Linux系统中关于网络管理的内容。

17.3.1 网络配置

对于操作系统来说，网络配置一般包括IP地址、子网掩码、默认网关、路由等基本信息。在本小节中，仅介绍基本信息的配置和ftp服务器的搭建。

对于IP地址、子网掩码、默认网关这些基本信息来说，可以使用`ifconfig`来实现，而配置路由信息则需要使用`route`来实现。这两个命令的基本使用方法在前面的章节中已经介绍过，在此将直接对这些信息进行配置，配置脚本如17-11.sh所示。

```
#示例脚本17-11.sh 网络基本配置，配置IP地址、子网掩码、默认网关和路由
#!/bin/bash
```

```
echo 使用ifconfig命令进行网络设置  
ifconfig eth0 192.168.1.1  
ifconfig eth0 netmask 255.255.255.0  
route add default gw 192.168.1.1
```

```
echo 显示最新的网络配置
```

```
ifconfig
```

为示例脚本17-11.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$ chmod u+x 17-11.sh  
ben@ben-laptop:~$ ./17-11.sh  
使用ifconfig命令进行网络设置 显示最新的网络配置  
eth0      Link encap:以太网  硬件地址 1c:c1:de:94:54:69  
          inet 地址:192.168.1.123  广播:192.168.1.255  掩码:255.  
          UP BROADCAST MULTICAST MTU:1500 跳点数:1  
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0  
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0  
          碰撞:0 发送队列长度:1000  
          接收字节:0 (0.0 B)  发送字节:0 (0.0 B)  
          中断:28  
lo       Link encap:本地环回  
          inet 地址:127.0.0.1  掩码:255.0.0.0  
          inet6 地址: ::1/128 Scope:Host  
          UP LOOPBACK RUNNING MTU:16436 跳点数:1  
          接收数据包:78 错误:0 丢弃:0 过载:0 帧数:0
```

```
发送数据包:78 错误:0 丢弃:0 过载:0 载波:0  
碰撞:0 发送队列长度:0  
接收字节:8924 (8.9 KB)  发送字节:8924 (8.9 KB)
```

在脚本17-11.sh中，对网卡设置了IP地址、子网掩码和路由信息等信息，这些构成了基本的网络配置信息。在运行该脚本之前，需要根据实际情况进行适当的修改，需要将eth0换成实际使用的网卡符号，而IP地址、子网掩码等信息也需要按照实际需要进行配置。当对网卡进行配置之后，就需要重启网卡，以使配置的内容生效。在Linux系统中，网卡的启动与关闭一般使用以下两个命令：

- **ifup eth0**: 启动eth0网卡。
- **ifdown eth0**: 关闭eth0网卡。

上面的两个命令分别实现了网卡的启动和关闭，对于一般的网卡来说，其重启的过程通常是先关闭网卡，然后再启动网卡，从而使新配置的内容生效。除此之外，还可以使用以下语句实现网卡的重启：

```
/etc/rc.d/init.d/network restart
```

一般来说，使用ifconfig命令进行网络设置之后，可以不必重启网卡，最新的配置信息就会生效。

注意

进行网络配置时，需要使用**root**用户或具有**root**权限，所以在执行示例脚本17-11.sh时，要使用**sudo**命令。

在Linux系统中，除了使用命令之外，还可以直接操作配置文件来完成网络的配置。和网络相关的配置文件如表17-2所示。

表17-2 网络相关配置文件

文件名称	作用
/etc/resolv.conf	设定DNS
/etc/services	定义一些服务端口
/etc/hostname	存储主机名
/etc/host.conf	确定主机名解释顺序
/etc/hosts	IP地址和主机名的映射

表17-2展示了部分与网络配置相关的文件，/etc/hostname文件中存储了主机名，主机名用来标识该主机。除了有主机名之外，还存储了和主机相关的完整域名信息，该文件的内容如下：

```
ben@ben-laptop:/etc$ cat /etc/hostname  
ben-laptop
```

配置文件/etc/hosts中存储着IP地址和主机名的映射关系，还包括主机的别名等信息，主要用来标识IP地址和主机名之间的关系，因为IP地址虽然用来标记网络地址，但是却难于记忆，使用主机名虽然易于记忆，但是不能直接指向网络地址。该文件的内容如下：

```
ben@ben-laptop:/etc$ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 ben-laptop

# The following lines are desirable for IPv6 capable hosts
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
```

文件/etc/services包含服务名和端口号之间的映射关系，该文件包含系统中安装的服务和使用的端口信息，其一般格式如下：

服务名	端口号	端口类型
-----	-----	------

该文件的部分信息如下：

ftp	21/tcp		
ssh	22/udp		
telnet	23/tcp		
smtp	25/tcp	mail	
login	513/tcp		
who	513/udp	whod	
shell	514/tcp	cmd	# no passwords used

syslog 514/udp

/etc/resolv.conf存放着系统的DNS，该文件中基本内容如下：

```
ben@ben-laptop:/etc$ cat /etc/resolv.conf
# Generated by NetworkManager
nameserver 192.168.0.1
```

注意

随着对Linux系统的操作的进行，配置文件的内容不会一成不变，而会随着用户的操作发生变化。

除了使用命令和修改配置文件对网络模块进行配置之外，Linux系统还提供了图形化的操作方式对网络进行配置。在任务栏中右击，打开网络连接中的【编辑连接】菜单选项，打开的网络连接配置界面如图17-4所示。



图17-4 【网络连接】界面

通过图17-4可以看出，在图形界面给出的操作环境中，可以实现对有线网络、无线网络、移动宽带、VPN以及DSL的配置。而最常用的就是对有线网络和无线网络进行各种配置。对于无线网络的配置，需要首先选中待配置的网卡，单击【编辑】按钮后就可以对无线网络进行配置了，无线网络的配置界面如图17-5所示。



图17-5 无线网络配置

在图17-5中，一般对IPv4进行配置，可以在【方法】中选择自动配置、手动配置等不同的配置方式。当选择自动（DHCP）方式时，系统

将自动获取IP地址和其他网络信息。当选择【手动】方式时，配置方式如图17-6所示。



图17-6 手动配置网络信息

在相应的位置输入需要配置的信息，就可以完成IP地址、子网掩码、网关以及DNS服务器的设置。在设定完之后，可以单击【应用】按钮来保存配置信息。如果当前的用户不是root用户或不具有root权限，那么还需要输入root密码来使配置信息生效，如图17-7所示。



图17-7 认证网络配置信息

当单击【授权】按钮后，配置工作就会最终完成。完成配置之后，需要重启网卡，从而使得配置信息生效。

17.3.2 服务器的安装

在Linux系统中，存在许多常用的服务器，如ftp服务器、nfs服务器、Samba服务器、Apache服务器等。这些服务器应用于不同的场合，如表17-3所示。

表17-3 常用服务器介绍

服务器名称	说明
ftp	最早实现文件共享和传输，支持FTP协议
nfs	数据网络文件系统，允许一个系统在网络上与他人共享目录和文件
Samba	实现Linux和Windows系统的文件和资源的共享
Apache	当前使用最广泛的Web服务器

表17-3中列出的这些服务器在Linux系统中会经常使用，在本小节中，仅介绍ftp服务器的安装与配置方式，其他服务器的配置方式与ftp服务器类似，读者仅需要将相应的软件包安装好，按照实际的需要对配置文件进行适当修改即可。

ftp服务器常用的是vsftpd服务器，系统默认是不安装该服务器的。vsftpd服务器从开始安装到最终使用的流程如图17-8所示。

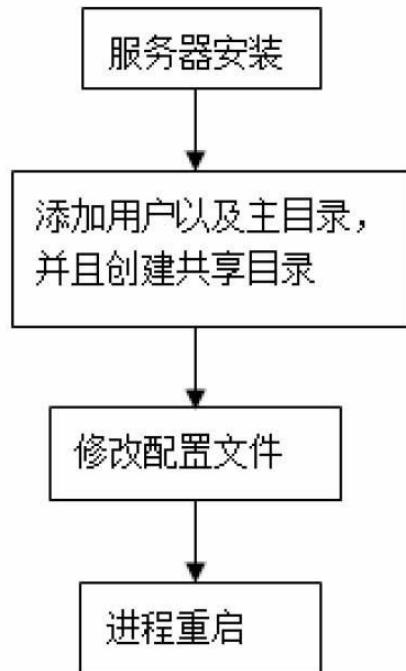


图17-8 vsftpd服务器使用流程

对于ftp服务器来说，在使用之前首先需要进行安装。而对于Ubuntu系统来说，可以使用apt-get命令实现，如图17-9 所示。

```

ben@ben-laptop:~$ sudo apt-get install vsftpd
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列【新】软件包将被安装：
  vsftpd

升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 316 个软件包未被升
级。
需要下载 141kB 的软件包。
解压缩后会消耗掉 471kB 的额外空间。
获取：1 http://cn.archive.ubuntu.com/ubuntu/ lucid-updates/main vsftpd 2.2.2-3ub
untu6.3 [141kB]
下载 141kB, 耗时 2 秒 (54.7kB/s)
正在预设定软件包 ...
选中了曾被取消选择的软件包 vsftpd。
(正在读取数据库 ... 系统当前总共安装有 163042 个文件和目录。)
正在解压缩 vsftpd (从 .../vsftpd_2.2.2-3ubuntu6.3_i386.deb) ...
正在处理用于 man-db 的触发器 ...
正在处理用于 ureadahead 的触发器 ...
ureadahead will be reprofiled on next reboot
正在设置 vsftpd (2.2.2-3ubuntu6.3) ...
正在将用户“ftp”加入到“ftp”组中
vsftpd start/running, process 2538

```

图17-9 vsftpd服务器安装过程

在安装服务器之后，理论上就可以使用了。但是为了便于使用，还需要指定共享目录文件，并且对配置文件进行必要的修改。指定共享目录文件的目的是使得服务器的目录能够统一，使操作比较方便。

对于vsftpd服务器来说，相对重要的是对配置文件进行修改，配置文件的内容如表17-4所示。

表17-4 vsftpd服务器配置文件常用配置项

配置项	作用	默认选项
listen	是否以独立模式运 行	NO
anonymous_enable	是否允许匿名登录	NO
local_enable	是否允许本地用户 登录	YES

write_enable	对服务器是否可写	YES
local_umask	本地掩码	022
anon_upload_enable	是否允许匿名上传	YES
anon_mkdir_write_enable	是否允许匿名创建目录文件	YES
xferlog_file	日志文件目录	/var/log/vsftpd.log
use_localtime	使用本地时间	YES
connect_from_port_20	是否通过端口20进行连接	YES
idle_session_timeout	空闲时间间隔	600, 单位: 秒
data_connection_timeout	连接时间间隔	120, 单位: 秒

表17-4列出了配置文件中的常用选项，除此之外，该配置文件中还有其他选项，可以根据实际需要进行相应的修改。

注意

文件中存在很多的以“#”开头的部分，这些是文件中的解释部分，在读取配置文件时，会略过这些内容。

对于该配置文件来说，比较常用的内容是是否独立模式、允许的最大连接数、是否允许匿名用户访问等。对这些内容就行修改以后，还需要重启服务，从而使得修改的内容生效，重启服务的命令如下：

```
/etc/init.d/vsftpd restart
```

除了使用restart命令之外，还可以使用start和stop命令来实现restart命令的效果。start和stop命令的使用方式如下：

- /etc/init.d/vsftpd stop
- /etc/init.d/vsftpd start

上面的两个命令中，stop用来实现停止ftp服务，在停止服务之后，用户就不能再访问ftp服务器，也不能进行任何操作了。只有使用start命令启动服务之后，ftp服务器才能正常使用。

17.4 日志管理

平时很多人都有写日记的习惯，用点滴笔墨记录自己的日常生活，从而在以后还能够回忆起快乐的时光。对于计算机操作系统来说，也需要写日志。这些日志记录着操作系统运行的诸多信息，能够方便用户通过日志来查看系统，从而进一步掌握系统的运行信息。本章将介绍Linux系统中日志的基本信息以及日志的备份和删除等基本操作。

17.4.1 日志简介

在Linux系统中，日志一般记录在日志文件中，所有的日志文件都放在目录/var/log中，日志的内容记录了操作系统的操作过程、系统运行信息、安全信息以及错误信息等。所有的日志文件都采用标准的日志文

件格式，并且大部分的日志文件是由系统日志守护进程syslogd根据系统和应用的配置要求生成的，如果系统或应用不需要记录日志，那么在日志系统中将不存在相关的日志。

在Linux系统中，所有的内容都被看做是文件，和日志相关的文件如表17-5所示。

表17-5 和日志相关的文件

文件名	绝对路径	作用
Message	/var/log/message	消息日志文件
Secure	/var/log/secure	安全相关的日志信息
maillog	/var/log/maillog	与邮件相关的日志信息
cron	/var/log/cron	与定时任务相关的日志信息
syslog	/var/log/syslog	系统日志文件
boot.log	/var/log/boot.log	守护进程启动和停止相关的日志消息

不同的日志文件中记录了系统不同的日志信息。Messages记录了系统中应用程序的一些基本操作信息，syslog日志文件属于系统日志文件，记录了系统的日常基本信息，而cron文件记录了与定时任务相关的信息。日志文件中的内容都是ASCII文本信息，可以使用文本操作的相关命令来查看日志信息。关于日志的基本操作如示例脚本17-12.sh所示。

```
#示例脚本17-12.sh 显示日志文件  
#! /bin/bash  
  
cd /var/log  
echo 显示所有系统日志文件  
ls | grep syslog
```

为示例脚本17-12.sh赋予可执行权限后执行脚本，结果如下：

```
显示所有系统日志文件  
syslog  
syslog.1  
syslog.2.gz  
syslog.3.gz  
syslog.4.gz  
syslog.5.gz  
syslog.6.gz  
syslog.7.gz
```

通过示例脚本17-12.sh的执行结果可以看出，在目录/var/log中存储了所有日志相关的文件，用户可以根据自己的需要，在该目录中查找日志文件。

17.4.2 守护进程syslogd

Linux系统中的日志信息大部分都是由守护进程syslogd产生的，syslogd进程记录了系统运行的所有基本信息。当系统内核或相关应用程序产生信息时，syslogd进程能够根据其配置文件/etc/syslog.conf中的配置信息，来决定对该信息进行何种处理方式。守护进程syslogd的配置文件/etc/rsyslog.conf的内容如下：

```
ben@ben-laptop:~$ cat /etc/rsyslog.conf
# /etc/rsyslog.conf Configuration file for rsyslog.

#
#
# For more information see
#
/usr/share/doc/rsyslog-doc/html/rsyslog_conf.html
#
# Default logging rules can be found in /etc/rsyslog.d/50-default.conf

#####
##### MODULES #####
#####

$ModLoad imuxsock # provides support for local system logging
$ModLoad imklog   # provides kernel logging support (previous versions)
#$ModLoad immark  # provides --MARK-- message capability
```

```
$KLogPath /proc/kmsg

# provides UDP syslog reception
#$ModLoad imudp
#$UDPServerRun 514

# provides TCP syslog reception
#$ModLoad imtcp
#$InputTCPServerRun 514

#####
##### GLOBAL DIRECTIVES #####
#####

#
# Use traditional timestamp format.
# To enable high precision timestamps, comment out the following line.
#
$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat

# Filter duplicated messages
$RepeatedMsgReduction on

#
# Set the default permissions for all log files.
#
```

```
$FileOwner syslog  
$FileGroup adm  
$FileCreateMode 0640  
$DirCreateMode 0755  
$Umask 0022  
$PrivDropToUser syslog  
$PrivDropToGroup syslog  
  
#  
# Include all config files in /etc/rsyslog.d/  
#  
$IncludeConfig /etc/rsyslog.d/*.conf
```

配置文件的基本格式如下：

[消息类型][处理方案]

在配置文件中，存在大量提示性信息，这部分信息都是注释性的内容。文件中的主要部分由消息类型和处理方案组成，在消息类型和处理方案之间需要使用TAB键进行分隔，而消息类型又包括facility和level两部分，这两部分之间使用点号“.”隔开，可以使用通配符“*”来表示该部分所有的内容都是用同样的操作方式。facility表示消息的类型，而level表示消息的重要级别。常用的消息类型如表17-6所示。

表17-6 常用消息类型

消息类型	取值	作用
------	----	----

kern	0	内核日志信息
user	1	用户日志信息
mail	2	邮件系统日志信息
daemon	3	系统守护进程日志信息
auth	4	安全管理日志信息
syslog	5	syslogd守护进程日志信息
lpr	6	打印服务日志信息
news	7	新闻组服务日志信息
uucp	8	uucp系统日志信息
cron	9	守护进程cron的日志信息
authpriv	10	私有的安全管理日志信息
ftp	11	ftp守护进程日志信息
空	12~15	系统保留
local0~local7	16~23	本地使用日志信息

消息级别表示了消息发生后的重要程度，如果级别很高，那么就必须立即处理，否则就会影响系统的正常运行。消息级别的取值如表17-7

所示。

表17-7 常用消息级别

消息级别	取值	作用
emerg	0	系统及错误，系统出现严重问题
alert	1	警告事件，必须立即采取纠正措施
crit	2	关键性事件
err	3	错误信息
warning	4	警告信息
notice	5	需要特别注意的信息
info	6	普通提示性信息
debug	7	调试信息

在配置文件中，可以使用通配符“*”来表示对所有的类型都采用同样的处理方式，如下所示：

```
news.* /var/log/warning  
*.err  
/var/log/err
```

对于上面的两个表达式来说，对类型为消息的日志信息，不论是什

么级别的，都要写入文件/var/log/warning中。而对于任何类型的错误信息来说，都需要写入/var/log/err文件中。

注意

对于不同的Linux系统版本来说，其守护进程的配置文件名也不相同。对于Red Hat来说，其守护进程的文件名为/etc/syslog.conf。

17.4.3 日志的备份操作

在Linux系统使用过程中，日志需要经常备份，从而保证一段时间内的日志信息不会丢失，从而方便用户后期在查找时能够找到相关记录。在Linux系统中，备份操作一般使用tar命令来对文件进行打包，然后将打包的tar包转移到其他目录中，从而备份日志信息。

tar命令的基本格式如下：

tar 选项 操作文件

tar命令有丰富的选项，而一般来说，tar命令还可以分为主选项和辅助选项，主选项是必须要有的选项，辅助选项可以根据实际需要决定是否选用。tar命令常用的选项如表17-8所示。

表17-8 tar命令常用选项

选项	70作用
-x	拆包命令，从tar包中将文件释放出来
-c	打包命令，将文件打包
-t	列出包中的文件
-v	在打包过程中显示文件
-z	是否具有gzip属性
-f	使用文件包名
-p	使用原文件的属性

tar命令在使用时主选项只能选择一个，而辅助选项可以使用一个，也可以同时使用几个。使用tar命令来备份日志文件的方法如示例脚本17-13.sh所示。

```
#示例脚本17-13.sh 使用tar备份日志
#!/bin/bash

echo进入日志目录
cd /var/log

echo对日志文件进行打包
tar -cvf * log.tar
ls -l | grep log.tar
echo
```

```
echo移动日志文件到备份目录
```

```
sudo mv log.tar /home
```

```
sudo rm log.tar
```

为示例脚本17-13.sh赋予可执行权限后执行脚本，结果如下：

```
进入日志目录
```

```
对日志文件进行打包
```

```
-rw-r--r-- 1 root          root 3635200 2014-03-31 22:29
```

```
移动日志文件到备份目录
```

```
-rw-r--r-- 1 root root 3635200 2014-03-31 22:29 log.tar
```

通过示例脚本17-13.sh的执行结果可以看出，在对日志文件进行打包以后，可以将日志文件移动到任何备份目录中，也可以存储到其他存储介质中，从而保存日志文件。

注意

tar命令只负责将文件打包，而不会对文件进行压缩。打包以后，文件的大小等于打包的文件的总和。

在Linux系统中还可以使用其他方式对日志进行备份，本章不再细述，有兴趣的读者可以参考其他资料。

17.4.4 日志的定时操作

对于日志的操作一般使用定时操作，并且需要周期性的操作，因此要使用cron定时任务来设定。在设定定时任务时，一般需要每天都备份当天的日志信息，在一段时间之后，确定日志信息没有使用的可能后，可删除没有使用价值的最早日志信息。删除日志信息可以使用rm命令，而备份日志一般使用tar命令，日志的定时操作实例如示例脚本17-14.sh所示。

```
#示例脚本17-14.sh 日志的定时操作
#!/bin/bash

echo 在午夜进行日志的备份操作
at -f 17-13.sh midnight
```

为示例脚本17-14.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~$chmod u+x 17-13.sh
ben@ben-laptop:~$./17-13.sh
在午夜进行日志的备份操作
warning: commands will be executed using /bin/sh
job 1 at Wed Jul 30 00:00:00 2014
```

通过示例脚本17-14.sh的执行结果可以看出，在午夜会按照示例脚本17-13.sh中的内容对日志进行备份操作。这种执行方式只可以执行一次，如果确实需要多次执行，可以对cron表格进行编辑，将该任务添加

到cron表格中，从而实现每天午夜都可以对日志进行处理。

17.5 小结

本章主要介绍了如何使用Shell脚本实现一些日常中常用的操作，如基本的系统检测、计划任务的实现、网络管理以及日志管理等。

对于Linux系统来说，各种资源都是写入相应文件中的，并且使用适当的命令就能获取需要的系统资源参数。

对于那些需要周期性执行的程序来说，at和cron是不可或缺的选择。at命令能够指定某个程序、脚本在规定的时间执行，并且只能执行一次。当下次需要执行时，还需要指定。而使用crom就可以使得任务能够周期性地执行，而不需要进行人工干预。用户可以根据实际需要选择使用哪种操作方式。

对于Linux来说，网络管理是系统中不可缺少的重要部分，而对于网络的管理也需要格外重视。可以使用特殊命令来实现网络配置、网卡流量检测等基本操作，从而使读者加深对网络模块的了解。

对于任何系统来说，日志系统用来记录系统中每一天的操作。在Linux系统中，日志是由守护进程syslogd来记录并产生的，所有的日志都记录在目录文件/var/log中。随着时间的推移，日志信息可能会不断增多，这就需要将日志进行备份处理，对很久以前的日志信息还可以有选择地删除。可以使用tar命令实现简单的日志备份，而使用rm命令可以将日志删除。

Linux系统的管理和基本操作还包括很多其他方面，而使用Shell脚

本也可实现相应的功能。有兴趣的读者可以进行深层次的研究。

第18章 Shell实战之数据库操作

现在的社会是一个信息大爆炸的社会，在日常生活中，每一个人的周围都充斥着大量数据。简单地说，每个人的手机中都储存着很多朋友的联系方式，这些联系方式包括移动电话、固定电话、地址、邮箱等。这些东西都需要储存在一个“仓库”中，从而在使用的时候能够到“仓库”中获取相应的信息，这个仓库就是数据库。数据库的操作除了使用图形化的操作环境以外，还可以使用Shell脚本。本章将重点介绍在Shell脚本中如何操作数据库。

本章的主要内容如下：

- Linux系统中的基本数据库：SQLite、MySQL介绍
- SQL语言基本介绍，以及如何在Shell中执行SQL语句
- 图书管理系统的实例操作

18.1 Linux系统中的数据库

不管是Windows系统，还是Linux系统，都可以使用数据库来存储数据。在Linux系统中，在数据量不是很大的情况下，使用数据库时一般会选择SQLite。而对于一般的需求来说，MySQL已经足够了。对于大型系统，一般会将Oracle作为首选数据库。本章只介绍常用的面向较小数量级的SQLite数据库和MySQL数据库。

18.1.1 SQLite简介

SQLite是一款非常轻巧的数据库，占用资源非常小，一般使用少量内存就可以流畅地运行，并且能够在各种主流操作系统（如Windows、Linux等）中运行，还提供了各种编程语言的接口来实现对SQLite的操作。

SQLite支持跨平台操作，并且其操作非常简单。除了使用命令进行操作之外，SQLite也有可以下载来自第三方SQLite的GUI软件。在默认的Ubuntu系统中，是不存在SQLite数据库的操作环境的，需要安装。在Ubuntu系统中安装SQLite可以使用apt-get命令，也可以通过Ubuntu软件中心进行安装。本小节将介绍如何使用Ubuntu软件中心进行安装。

Ubuntu软件中心是Ubuntu系统中提供的一个软件包管理工具，该工具可以非常简单地实现系统中软件的安装、卸载、升级等工作。在【应用程序】中打开【Ubuntu软件中心】，如图18-1所示。



图18-1 【Ubuntu软件中心】界面

在打开的【Ubuntu软件中心】中，选择要添加的软件，然后单击右边的【安装】按钮，系统就自动安装选中的软件。如果不知道需要安装

的软件是否存在，可以在右上角的窗口中输入软件名称，然后进行查询，在查询出来的结果中选择合适的软件进行安装，如图18-2所示。

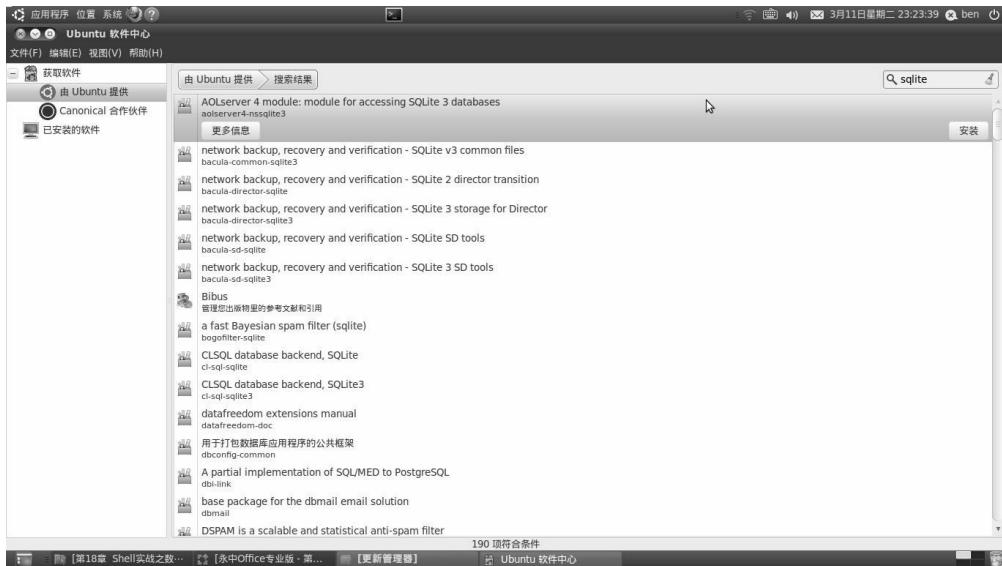


图18-2 选择合适的软件

在Ubuntu软件中心中可以查询已经安装的软件，选中【已安装的软件】以后，就能显示当前系统中已经安装的软件，如图18-3所示。



图18-3 显示已安装的软件

如果某个已安装软件不再使用，可以卸载它。在界面中选中需要卸载的软件，在该软件的右边会显示一个【卸载】按钮，单击【卸载】按钮，就能将该软件从系统中卸载，如图18-4所示。

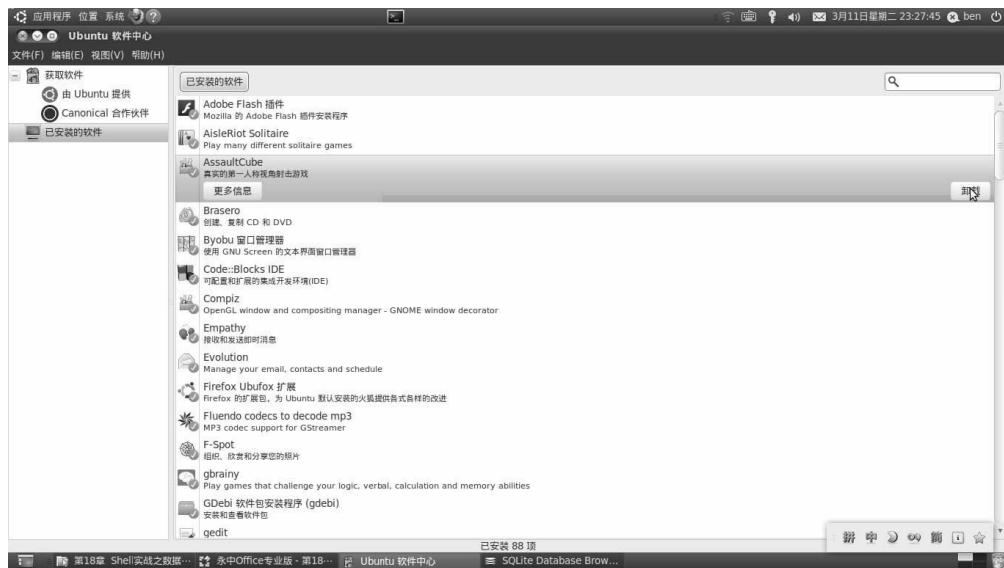


图18-4 卸载软件

上面以SQLite数据库的安装介绍了如何利用Ubuntu软件中心安装软件，使用Ubuntu软件中心可以方便地进行软件的安装、卸载等管理工作。

注意

在Ubuntu软件中心展示的软件种类来自于配置文件/etc/apt/sources.list，如果发现需要的文件不存在，可以更新该文件来获取需要的软件。

18.1.2 SQLite的图形化操作

为了操作方便，SQLite数据库还提供了图形化的操作环境，这样就可以进一步简化数据库的操作。在Linux系统中，一般使用第三方软件作为SQLite的图形化操作环境。依次单击【应用程序】|【编程】|【SQLite】就可以打开SQLite的图形化操作工具，如图18-5所示。

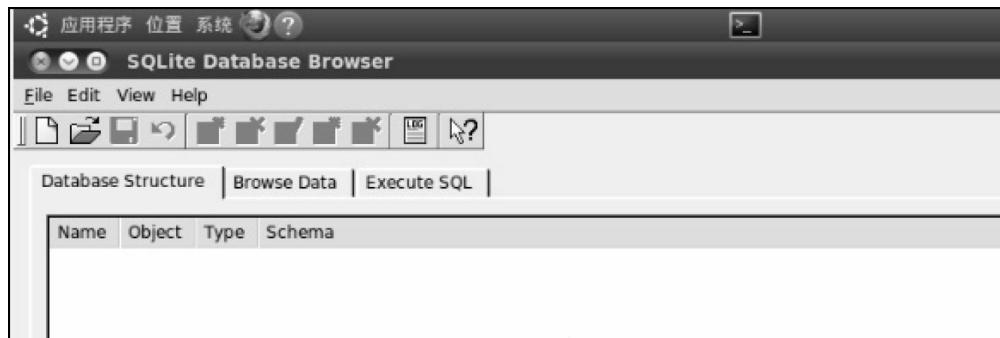


图18-5 SQLite图形化操作环境

通过图18-5可以看出，在SQLite的图形化操作环境中，可以进行各种操作来代替SQL语句的操作。在工具栏中，可以非常轻松地实现创建表、删除表、编辑表等各种操作，还可以查看数据表的数据结构、浏览表中的数据以及执行SQL语句。下面对常用的操作进行简单介绍。

在对数据库操作之前，需要首先创建数据库。可以选择如下两种方式进行创建：

- 依次打开【文件】|【创建新数据库】
- 单击工具栏中的【新建数据库】按钮

创建数据库时需要输入数据库的名称，然后选择文件的存储路径，最后单击【Save】按钮，对文件进行保存，如图18-6所示。

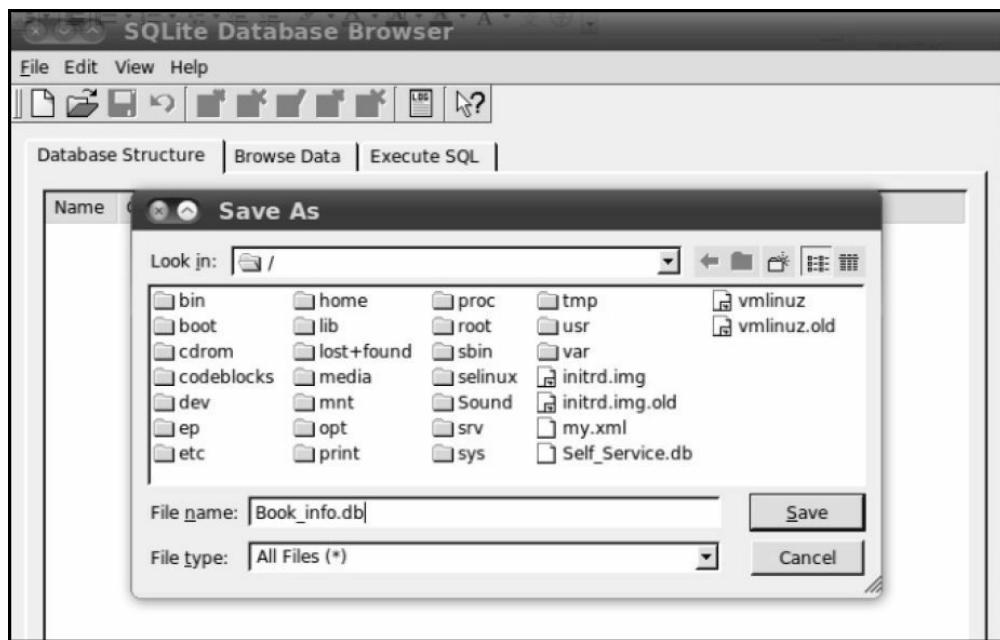


图18-6 创建数据库

在单击了【Save】按钮之后就完成了数据库的创建。如果数据库中没有数据表，那么就直接跳转到创建表的界面中，如图18-7所示。

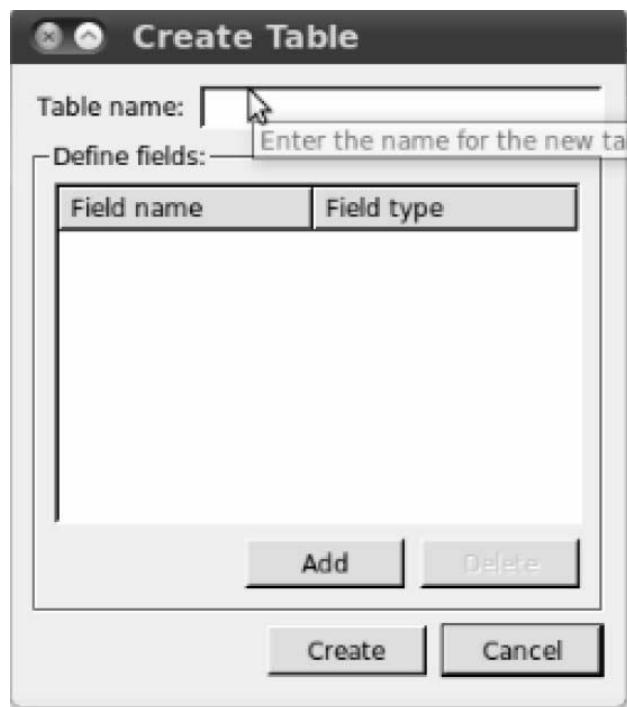


图18-7 创建数据表

创建表时需要编辑数据表名和添加数据列。在图18-7中鼠标的位置键入需要创建的数据表的名称，然后添加数据表中的各个字段。单击【Add】按钮，就可以实现数据列的添加，如图18-8所示。

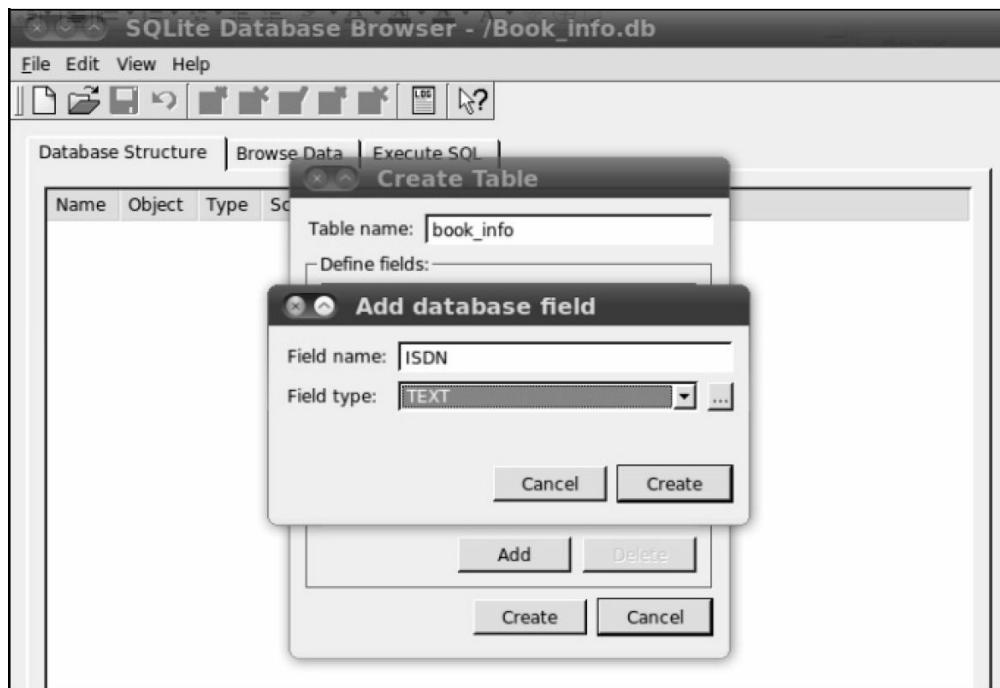


图18-8 为数据表添加字段

在添加数据列时，需要同时指定数据列的数据类型，如图18-8所示。当完成所有数据列的编辑时，单击【Create】按钮，就完成了数据表的添加。当所有的数据列都添加完毕后，单击【Create Table】页面中的【Create】按钮，就可完成整个数据表的添加。

当数据表创建成功之后，就可以向数据表中添加相应的数据了。在【Browse Data】标签页中通过单击按钮【New Record】就可以增加一行新的纪录，在对应的字段位置添加相应的数据内容，就完成了一条记录的添加。除了可以添加数据之外，还可以删除记录或查看表中存在哪些记录，如图18-9所示。

The screenshot shows the SQLite Database Browser interface with the title bar "SQLite Database Browser - /Book_info.db". The menu bar includes File, Edit, View, Help. Below the menu is a toolbar with icons for file operations like Open, Save, and Print. The main area has tabs: Database Structure, Browse Data, and Execute SQL. The "Browse Data" tab is selected, showing a table named "book_info". The table has four columns: book_author, book_code, book_type, and ISDN. There are three rows of data:

	book_author	book_code	book_type	ISDN
1	benben	1	computer	0000001
2	ben	2	computer	0000002
3	ben	3	Maths	0000003

Buttons for New Record and Delete Record are visible at the top right of the table view.

图18-9 添加数据示意图

在完成数据表结构的编辑之后，就可以查看数据表的结构以及数据表中的内容了，并且能够执行SQL语句。单击【Database Structure】标签页，就可以查看数据库中所有的表结构，单击相应的数据表前面的加号，就可以显示该数据表的数据结构，如图18-10所示。

The screenshot shows the SQLite Database Browser interface with the title bar "SQLite Database Browser - /Book_info.db". The menu bar includes File, Edit, View, Help. Below the menu is a toolbar with icons for file operations like Open, Save, and Print. The main area has tabs: Database Structure, Browse Data, and Execute SQL. The "Database Structure" tab is selected. It displays a table of objects in the database. The "book_info" table is expanded, showing its schema:

Name	Object	Type	Schema
book_info	table		CREATE TABLE book_info (book_author TEXT, book_code NUMERIC...
	book_author	field	TEXT
	book_code	field	NUMERIC
	book_type	field	TEXT
	ISDN	field	TEXT

图18-10 查看数据表结构

在SqlLite中还可以执行SQL语句。单击【Execute SQL】标签页，可以在【SQL string】窗口内添加SQL语句，单击【Execute query】按钮后，SQL语句就被执行。执行的结果将显示在【Data returned】窗口中，如图18-11所示。

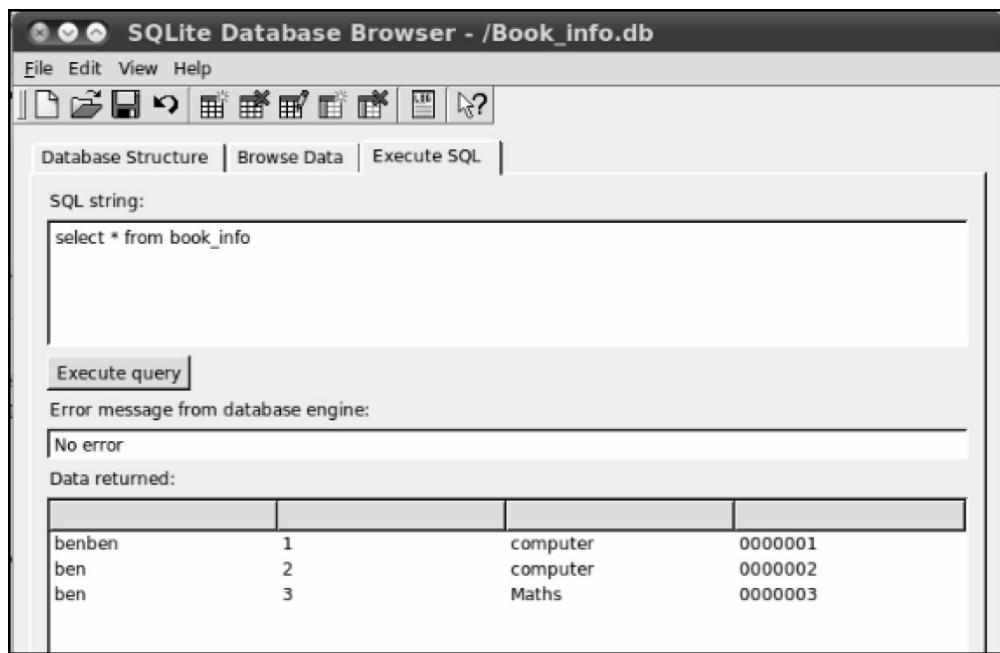


图18-11 SQL语句操作

如果执行的SQL语句出现错误，那么在中间的输出框中会显示错误的详细信息。使用者可以根据展示的错误信息对语句进行必要的修改，从而保证SQL语句能够正常执行，显示需要的结果。

注意

在对SQLite进行操作时，建议使用root用户操作。否则有可能会由于权限不足而不允许进行操作。

18.1.3 MySQL简介

MySQL是一个关系型数据库管理系统，并且是一种关联数据库管理系统。所谓关联数据库就是将不同的数据按照一定的关系保存在不同

的表中，而不是将所有数据放在同一个表内，这样就增加了数据操作的速度和灵活性。MySQL最大的特点是该数据库是开源的，其源代码全部对外开放，可以大大降低开发成本，从而使得一些中小型项目都将MySQL作为首选数据库，并且可以搭配PHP、Apache一起使用，和Linux系统一起被称为“LAMP组合”。

注意

LAMP组合是指在Linux操作系统中，使用Apache作为Web服务器，MySQL作为数据库，而PHP作为服务器端的脚本解释器。

MySQL全部代码都使用C/C++编写，并且使用多种编译器进行测试，保证源代码能够在各种平台上顺利运行。MySQL还支持多种操作系统，如Linux、Windows、AIX系统等。对于每一种操作系统都提供了很多API（应用程序接口），在系统中可以使用多种语言来实现对MySQL的各种操作。

在对数据库进行操作时，要保证服务器是启动状态。如果服务器未启动，需要首先启动服务器。在Linux系统中，启动相应的服务器需要使用相应的命令，如启动MySQL服务器，就使用命令MySQL，如果启动SQLite服务器，就使用SQLite命令。在启动服务器时，需要确定服务可执行程序的安装位置，如果在系统中找不到对应的命令和安装程序，那么需要首先安装相应的服务器。

在Ubuntu系统中，可以使用apt-get命令来进行安装，还可以在安装中心中进行安装。使用apt-get命令安装MySQL服务器的方式如图18-12

所示。

```
ben@ben-laptop:~$ sudo apt-get install mysql-server mysql-client
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会安装下列额外的软件包：
libdbd-mysql-perl libdbi-perl libhtml-template-perl libmysqlclient16
libnet-daemon-perl libplrpc-perl mysql-client-5.1 mysql-client-core-5.1
mysql-common mysql-server-5.1 mysql-server-core-5.1
建议安装的软件包：
dbishell libipc-sharedcache-perl tinyca mailx
下列【新】软件包将被安装：
libdbd-mysql-perl libdbi-perl libhtml-template-perl libnet-daemon-perl
libplrpc-perl mysql-client mysql-client-5.1 mysql-client-core-5.1
mysql-server mysql-server-5.1 mysql-server-core-5.1
下列软件包将被升级：
libmysqlclient16 mysql-common
升级了 2 个软件包，新安装了 11 个软件包，要卸载 0 个软件包，有 311 个软件包未被
升级。
需要下载 23.3MB 的软件包。
解压缩后会消耗掉 50.6MB 的额外空间。
您希望继续执行吗？[Y/n] [
```

图18-12 获取MySQL的相关软件包

在获取到相应的安装信息后，需要用户选择是否继续执行，如果输入n，那么就会直接退出安装过程。如果确定安装，就选择y，这样系统就会自动安装MySQL服务器和客户端，如图18-13所示。

```
libplrpc-perl mysql-client mysql-client-5.1 mysql-client-core-5.1
mysql-server mysql-server-5.1 mysql-server-core-5.1
下列软件包将被升级：
libmysqlclient16 mysql-common
升级了 2 个软件包，新安装了 11 个软件包，要卸载 0 个软件包，有 311 个软件包未被
升级。
需要下载 23.3MB 的软件包。
解压缩后会消耗掉 50.6MB 的额外空间。
您希望继续执行吗？[Y/n]
获取：1 http://cn.archive.ubuntu.com/ubuntu/ lucid-updates/main mysql-common 5.1
.73-0ubuntu0.10.04.1 [72.6kB]
获取：2 http://cn.archive.ubuntu.com/ubuntu/ lucid/main libnet-daemon-perl 0.43-
1 [46.9kB]
获取：3 http://cn.archive.ubuntu.com/ubuntu/ lucid/main libplrpc-perl 0.2020-2 [
36.0kB]
获取：4 http://cn.archive.ubuntu.com/ubuntu/ lucid/main libdbi-perl 1.609-1build
1 [798kB]
获取：5 http://cn.archive.ubuntu.com/ubuntu/ lucid-updates/main libmysqlclient16
5.1.73-0ubuntu0.10.04.1 [1,902kB]
获取：6 http://cn.archive.ubuntu.com/ubuntu/ lucid/main libdbd-mysql-perl 4.012-
1ubuntu1 [135kB]
获取：7 http://cn.archive.ubuntu.com/ubuntu/ lucid-updates/main mysql-client-cor
e-5.1 5.1.73-0ubuntu0.10.04.1 [155kB]
13% [7 mysql-client-core-5.1 62.9kB/155kB 40%] 107kB/s 3分 9秒 [
```

图18-13 安装SQL服务器和客户端

图18-13表示的是正在下载相关的软件包，当软件包下载完成之后，会自动进行安装。在安装时，需要输入用户密码，如图18-14所示。

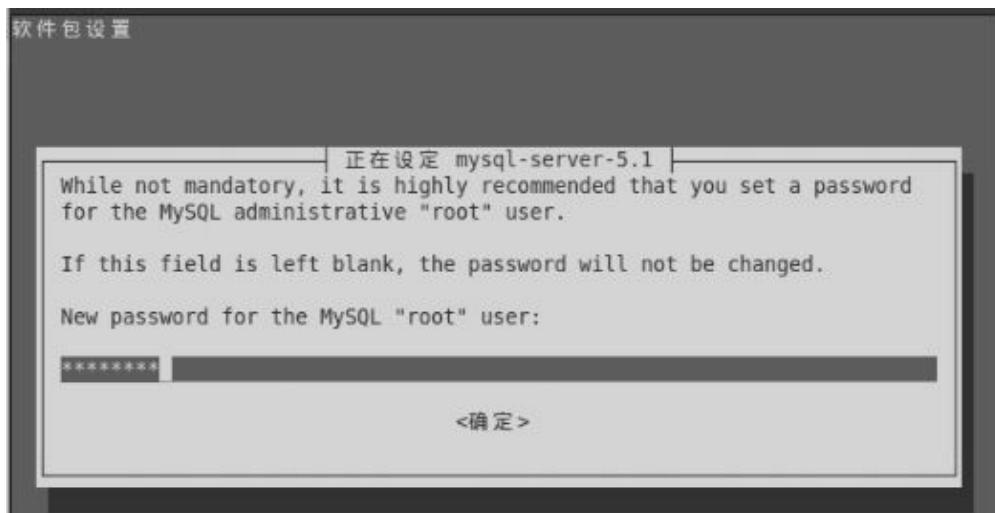


图18-14 输入MySQL管理员密码

在输入root用户的密码以后，单击【确定】按钮，然后MySQL就会自动被安装到系统中，安装完成后，MySQL就可以正常使用了，如图18-15所示。

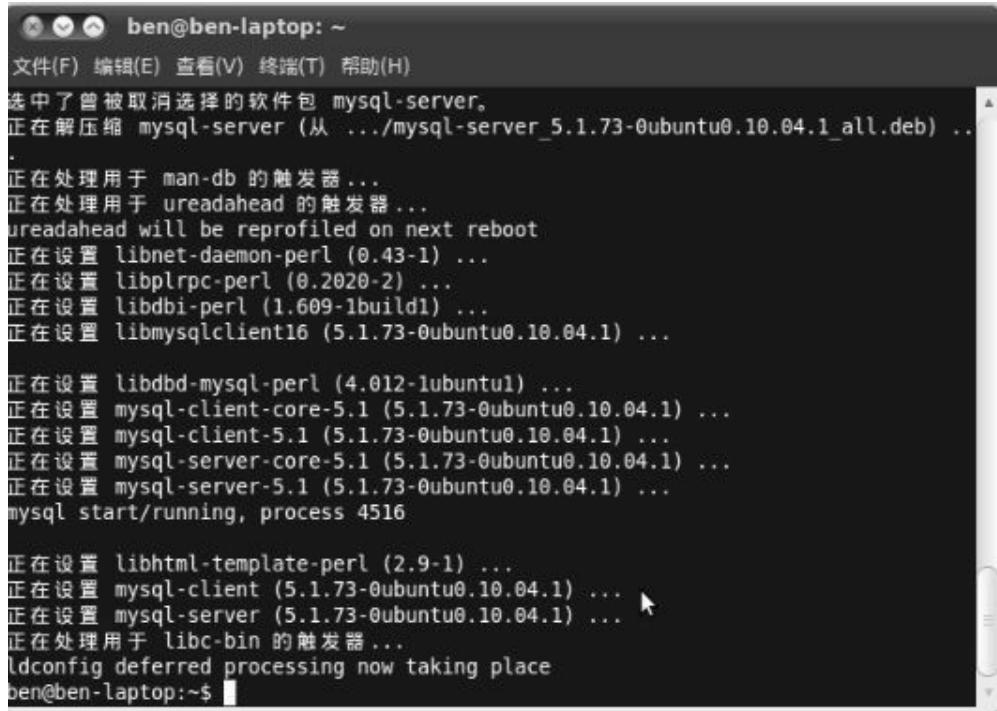


图18-15 MySQL软件包配置示意图

当MySQL安装完成之后，就可以使用相关的命令对MySQL进行各种操作了，如对数据库的操作以及对数据库中数据的操作。在进行操作时，可以直接使用MySQL命令进入MySQL的操作的命令行窗口，如下所示：

```
ben@ben-laptop:~ $mysql -u root -p  
Enter password:  
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 53  
Server version: 5.0.96-log Source distribution
```

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All

Oracle is a registered trademark of Oracle Corporation and/o

```
affiliates. Other names may be trademarks of their respective  
owners.
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql> use Book_Info;  
Database changed  
mysql> select * from book_info;
```

```
+----+-----+-----+-----+  
| ISDN | book_code | book_type | book_author |  
+----+-----+-----+-----+  
| 001 | 1     | computer   | benben    |  
| 002 | 2     | computer   | ben       |  
| 003 | 3     | maths      | ben       |  
+----+-----+-----+-----+  
2 row in set (0.00 sec)  
mysql>
```

通过上面的操作界面可以实现MySQL的几乎全部操作，用户可以选择在该命令行中进行操作。在执行SQL语句时，需要在语句的后面添加分号，用来标识语句的结束。

注意

在安装MySQL时，只有root用户才能进行安装，因此应该使用root用户进行安装或者使用sudo命令临时给普通用户安装，否则会提

示权限不足。

18.2 SQL语句

在操作数据库时，一般可以用SQL语句来实现数据库的操作。SQL语言作为现在关系数据库的通用语言而得到广泛使用。本节将重点介绍如何使用SQL语句来完成对数据库的各种操作。

18.2.1 SQL语言基本介绍

SQL语言是一种数据库操作语言，是结构化查询语言（Structured Query Language）的简称，是现在通用的一种数据库查询和程序设计语言，广泛应用于关系数据库系统的查询、修改、管理等基本操作。SQL语言中主要包含以下3部分：

- 数据定义（DDL）
- 数据操纵（DML）
- 数据控制（DCL）

这3部分涵盖了对数据库、表、表中的数据的各种操作，如创建表、对表中的数据进行增删改查等基本操作。SQL语句在使用过程中只需要告诉计算机要“做什么”，然后计算机就能够自动去执行。

SQL语言非常简洁，整个SQL语言本身只有不到100个单词，而经常使用的语句仅有6个，经过简单的学习就可以比较熟练地使用。它还

提供了数量众多、功能强大的函数，在使用过程中可以使用提供的函数来处理一些日常中经常遇到的问题。

18.2.2 基本的SQL操作

最基本的数据库的操作一般是增删改查，也就是在数据库中对记录进行增加、删除、修改和检索的基本操作。对于MySQL来说，可以使用标准SQL语句来进行操作。

1. 创建表

在SQLite中，可以使用如下语句来创建数据表：

```
create table userinfo(
    user_id    int,
    name       varchar(20),
    address    varchar(40),
    serial_number, varchar(16)
)
```

在上面的实例中，使用`create table`来创建数据表userinfo，在表中添加相应的字段，其基本格式如下：

字段名	字段的数据类型
-----	---------

数据表可以根据实际需要选择使用多少个字段，但是每个字段必须要添加数据类型（也可以称为数据约束），MySQL中的数据类型如表18-1所示。

表18-1 MySQL中使用的数据类型

数据类型	作用	数据类型	作用
int	整型	date	日期类型，格式为YYYY-MM-DD
char	描述固定长度的字符	time	时间类型，格式为HH:MM: SS
float	浮点型，即小数类型	timestamp	日期和时间类型
varchar	描述可变长度的字符串	text	长字符串类型
boolean	描述布尔值		

在创建数据表时，还可以使用primary key来设定数据表中的某个字段为主键，或者使用NOT NULL来设定该字段不能为空。设定为主键的字段不能为空，并且保证每一条记录的该字段的数值不能重复，如果设定为字段非空，那么该字段不能出现空值，如下列的SQL语句所示：

```
create table userinfo(
    user_id    int primary key,
    name       varchar(20),
```

```
address  varchar(40),  
serial_number, varchar(16) NOT NULL  
)
```

在上面的语句中，user_id字段设定为表userinfo的主键，因此在表userinfo中，不允许出现重复的user_id值。而字段serial_number设定为非空值，那么就不能出现空的serial_number字段。

2. 增加记录

在MySQL中，插入记录一般使用insert语句，该语句的一般形式如下：

```
insert into 表名[字段列表] values(数值列表)
```

在上面的基本形式中，字段列表是可选项，如果不添加字段列表，那么在进行数据的插入时，将按照为所有的字段赋值来处理。values后面的数值列表必须和字段列表一一对应，否则在插入的时候会提示数据类型不匹配。字段列表和数值列表都是通过逗号“，”进行分隔的。如果是字符串类型，还需要使用双引号括起来，insert语句的用法如下：

```
insert into userinfo(user_id, name, address, serial_number)  
insert into userinfo values(880009530002, "李四", "北京市",  
insert into userinfo(user_id, serial_number) values(880009530003, "王五", "上海市")
```

在上面的3条语句中，第1条是标准的使用方式，字段列表和数值列

表都按照创建数据表时的结构进行添加。第2条省略了字段列表，而数值列表按照创建数据表时的结构进行添加，并且是逐一对的。对于第3条来说，直接插入user_id和serial_number这两个字段，而对于可空字段name、address，使用默认为空的操作。

3. 查询记录

在MySQL中，使用select命令可以检索出表中符合条件的数据，select命令的基本形式如下：

```
select 字段列表 from 表名 [条件]
```

使用select命令时，需要查找哪个字段，就在字段列表中列出该字段。如果有多个字段，可以使用逗号将字段进行分隔。如果需要查询数据表中的所有数据，可以直接使用通配符“*”来表示。使用select命令时，可以添加条件，也可以不使用条件。当不使用条件时，默认查找该表中的所有记录。select命令的使用方式如下：

```
select * from userinfo;
select * from userinfo where user_id = '880009530002';
select * from userinfo where serial_number = '18600000003';
```

上面的3条语句中，第1条语句是查询userinfo表中所有的记录，而第2条语句和第3条语句分别查找user_id为880009530002和serial_number为18600000003的记录。如果在数据库中存在相应的记录，那么就会显示相应的信息，如果没有记录，那么就显示为空。

4. 修改记录

数据库中的记录可以被修改，在MySQL中可以使用update命令来进行修改，update命令的基本使用方式如下：

```
update 表名 set 字段名=新值 [条件]
```

使用update命令修改字段时，如果不使用条件，那么数据表中所有记录中的当前字段都会被修改为新值，当需要一次修改多个字段时，只需要在set命令后使用多个表达式来表示即可，而表达式之间需要使用逗号分隔，update命令的使用方式如下：

```
update userinfo set user_id='880009530004';  
update userinfo set user_id='880009530003' where serial_number=18600000003;
```

上面的语句中，第1条语句执行以后，数据表中所有的user_id都会变成880009530004，而当执行了第2条语句以后，serial_number为18600000003的user_id就变成了880009530003。

5. 删除记录

在MySQL中，删除记录可以使用delete命令实现。delete命令的基本使用如下：

```
delete from 表名 [条件]
```

在上面的表达式中，`delete`命令将删除表中符合条件的数据，如果不添加任何条件，将删除表中所有的数据。因此，在使用`delete`命令的时候要注意记录是不是真的需要删除，一旦删除以后，记录将很难恢复。`delete`语句的使用示例如下：

```
delete from userinfo where user_id = '880009530001';
select * from userinfo ;
delete from userinfo ;
select * from userinfo ;
```

注意

在`delete`命令的后面没有星号，只有在`select`命令后面才会出现星号。

上面介绍的是在MySQL中使用频率最高的几个语句，这些语句在其他的数据库系统中也能使用，这些语句同属于SQL中的内容，在所有支持SQL语句的数据库中都可以使用，如在SQLite、MySQL、Oracle等数据库中，都可以使用上面的语句来实现数据库的增删改查。

18.2.3 在Shell脚本中执行SQL语句

上面介绍了SQL语句的基本知识，掌握了上面的基本知识以后，就

可以在Shell脚本中使用这些语句来实现对数据库的操作了。

在确定安装好MySQL服务器之后，就可以使用MySQL命令来对数据库进行各种操作了。在对数据库操作之前，需要首先连接数据库。在连接数据库时，需要指定使用哪个用户连接哪个数据库，而在连接的时候还需要输入用户的密码来验证用户的有效性和合法性。运行MySQL命令之后，MySQL服务器就同时启动。在登录服务器时，需要指定数据库的有效用户以及对应的密码，可以在启动服务器时同时指定用户名和密码，一般使用-u选项指定使用的用户，而密码可以使用-p选项来指定。这种使用方式如下：

```
mysql -u root -p binghehj
```

在Shell脚本中，可以将需要执行的SQL语句放到MySQL的命令行中，从而使SQL语句能够被执行。对于MySQL来说，如果需要执行语句，那么需要使用-e选项来实现语句的添加。当MySQL命令执行完之后，相应的SQL语句也同时被执行，如示例脚本18-1.sh所示。

```
#示例脚本18-1.sh  mysql的基本使用
#!/bin/bash

echo 连接数据库
mysql -u root -p binghehj

echo 执行查询语句
mysql -e "select * from book_info"
```

为示例脚本18-1.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 18-1.sh
ben@ben-laptop:~ $./18-1.sh
连接数据库
执行查询语句
+----+-----+-----+-----+
| ISDN | book_code | book_type | book_author |
+----+-----+-----+-----+
| 001 | 1 | computer | benben |
| 002 | 2 | computer | ben |
| 003 | 3 | maths | ben |
+----+-----+-----+-----+
2 row in set (0.00 sec)
```

对于复杂的语句来说，如创建表、负责的查询语句等，还可以将所有的语句封装到EOF语句块中。EOF一般和符号“<<”一起使用，一般是两个EOF组成一个语句块，中间的语句作为子命令或子Shell的输入，执行完这些语句之后再返回到主调Shell。

18.3 图书管理系统中数据库操作实例

在前面的章节中介绍了如何在Shell脚本中使用SQL语句以及如何使用SQL语句来对数据库中的数据进行操作。本节将讲述如何在Shell脚本

中使用SQL语句来对具体的实例进行操作。下面仅以操作一条记录为例实现对数据库以及数据表中数据的各种操作。

18.3.1 数据库操作基本流程

对数据库操作的前提条件是打开数据库，也就是能够连接上数据库。在连接数据库时，需要指定连接哪台主机上的哪个数据库，还需要指定使用哪个用户连接，以及用户的登录密码。当连接上数据库以后，如果需要的数据表不存在，还需要创建这些表。在表创建完成后，就需要添加数据了。可以根据实际需要添加必要的数据。当数据需要修改时，可以更新表中的需要修改的字段。当数据不再使用时，还可以将数据进行删除。当数据库不再使用时，需要断开和数据库的连接。数据库操作的一般流程图如图18-16所示。

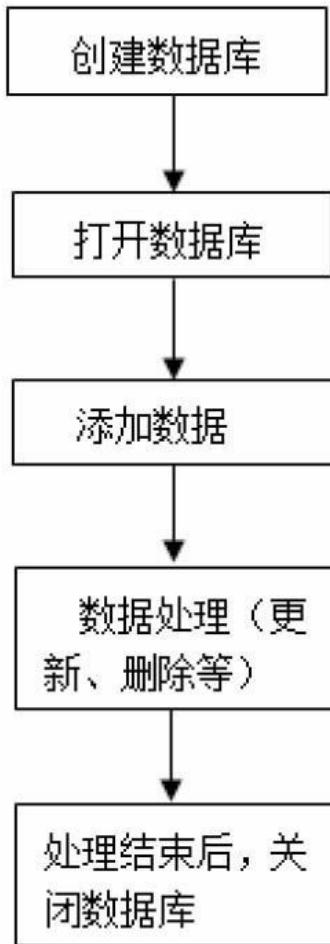


图18-16 数据库操作流程

18.3.2 创建表

数据库操作的第一步就是保证能够连接上数据库，并且将数据库打开。数据库连接的实例已由示例脚本18-1.sh展示了，读者可以参照它进行学习。

数据库中可创建各种需要的数据表，这些表用来存储数据。使用Shell脚本实现创建表的语句如下所示。

```
create table BookInfo
{
    ISBN varchar(20) NOT NULL,
    Book_Code nvarchar(100) NOT NULL,
    Book_Name nvarchar(100) ,
    Book_Type nvarchar(50) ,
    Publish_Name nvarchar(100) ,
    Author_Name nvarchar(100) ,
    Book_Price decimal(30, 2) ,
    Book_Quantity int ,
    Book_Page int ,
    Book_Num int ,
    Book_Desc nvarchar(max) ,
    Buy_Date date ,
    Is_User Boolean,
    Create_Date datetime
}
```

使用上面的语句就可以创建一个关于图书信息的表BookInfo，在表中存在描述图书的各种信息，其中，ISBN代表每一本书的国际标准编码ISBN，而Book_Code表示该书的一个编号，使用这两个编号来唯一地标识一本书。其余的信息都是可以为空的信息，如这本书的书名、类型、出版社、作者姓名、价格等。

在Shell脚本中调用上述语句就可以在数据库中创建表BookInfo，将上面的建表语句赋值给变量CreateTableSql，其调用方式如示例脚本18-2.sh所示。

```
#示例脚本18-2.sh 建表的基本使用
#!/bin/bash

echo 连接数据库
mysql -u root -p binghehj

echo 创建表
mysql -e "${CreateTableSql}"
```

为示例脚本18-2.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 18-2.sh
ben@ben-laptop:~ $./18-2.sh
连接数据库
创建表
```

通过示例脚本18-2.sh的执行结果可以看出，连接上数据库以后，就可以进行诸如创建表等各种操作了，而使用类似的语句可以创建各种需要的内容，能够连接上数据库是进行各种操作的前提。

18.3.3 增加图书信息

在创建了数据表之后，就可以向表中插入数据了。在插入数据时，一般使用SQL语言中的insert语句来完成。insert语句一般通过使用MySQL命令来实现数据的插入，即使用MySQL命令的-e选项来执行

insert语句，从而最终完成数据的插入语句。其使用方式如示例脚本18-3.sh所示。

```
#示例脚本18-3.sh 插入语句的基本使用
#!/bin/bash

InsertSql="INSERT INTO Book_Info
    (ISDN
     ,Book_Code
     ,Book_Name
     ,Book_Type
     ,Publish_Name
     ,Author_Name
     ,Book_Price
     ,Book_Quantity
     ,Book_Page
     ,Book_Num
     ,Book_Desc
     ,Buy_Date)
VALUES
    ('001'
     ,N'C++入门经典'
     ,N'教科书'
     ,N'清华大学出版社'
     ,N'黄静'
     ,121.50
```

```
, 2  
, 758  
, N'该书适合初学者，入门教程'  
, '2014-01-22');"  
echo连接数据库  
mysql -u root -p binghehj  
  
echo 插入记录  
mysql -e "${InsertSql}"
```

为示例脚本18-3.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 18-3.sh  
ben@ben-laptop:~ $./18-3.sh  
连接数据库  
插入记录
```

通过示例脚本18-3.sh可以在表中插入一条语句，可以使用多条insert语句完成多条记录的插入，从而将需要的数据添加到数据库中。

18.3.4 修改图书信息

如果图书的信息中间发生变化，如该图书借阅者发生变化、库存量发生变化、图书已经损坏等，此时就需要对图书的信息进行修改，从而得到最新的图书信息。可以使用update语句来实现图书信息的修改，如

示例脚本18-4.sh 所示。

```
#示例脚本18-4.sh 插入语句的基本使用
#!/bin/bash

UpdateSql="update Book_Info set Book_Num=10 where ISDN=001"
echo连接数据库
mysql -u root -p binghehj

echo 插入记录
mysql -e "${UpdateSql}"
```

为示例脚本18-4.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 18-4.sh
ben@ben-laptop:~ $./18-4.sh
连接数据库
插入记录
```

通过示例脚本18-4.sh可以更改数据库中ISDN为001的图书的库存数量，当更新完毕后，再次查询数据库，得到的概述的库存数量就变成了10，而不再是以前的数据了。

18.3.5 删 除 图 书 信 息

如果某本书不再使用，那么为了节省存储空间，就可以将该书相关的信息从数据库中删除，将数据从数据库中删除以后，再进行查询时，就找不到相关的信息了，如示例脚本18-5.sh所示。

```
#示例脚本18-5.sh    删除语句的基本使用
#!/bin/bash

DeleteSql="delete Book_Info where ISDN=001"
echo连接数据库
mysql -u root -p binghehj

echo 删除记录
mysql -e "${DeleteSql}"
```

为示例脚本18-5.sh赋予可执行权限后执行脚本，结果如下：

```
ben@ben-laptop:~ $chmod u+x 18-5.sh
ben@ben-laptop:~ $./18-5.sh
连接数据库
删除记录
```

通过示例脚本18-5.sh可以将数据库中ISDN为001的所有记录全部移除，再次查询数据库时，该类型的记录将不再存在。

注意

在对数据库中的数据进行更新、删除等操作时，需要仔细核实是

否需要删除这些记录，如果对数据进行了错误的处理，在一般情况下，是很难复原的。

18.4 小结

本章主要介绍了数据库的基本知识以及如何在Shell脚本中使用SQL语句对数据库进行操作。在Shell脚本中使用SQL语句可以非常方便地对SQLite和MySQL进行数据的各种操作。

数据库作为数据存储的方式之一，现在已经用于各种环境中。在使用了结构化查询语言SQL以后，就可以实现对多种关系型数据库的各种操作。SQLite作为一种“短小精悍”的数据库，广泛用于嵌入式系统中。而MySQL因为其源代码的开放特性，降低了软件开发的应用成本，进而增加了使用的范围。

SQL语句作为一种结构化查询语言而广泛用于大部分的关系型数据库，可以使用简单的语句实现数据的增删改查等基本操作。还可以将这些语句嵌入到Shell脚本中，从而完成自动化的数据库操作。